

P&S Heterogeneous Systems

Parallel Patterns: Histogram

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2022

14 November 2022

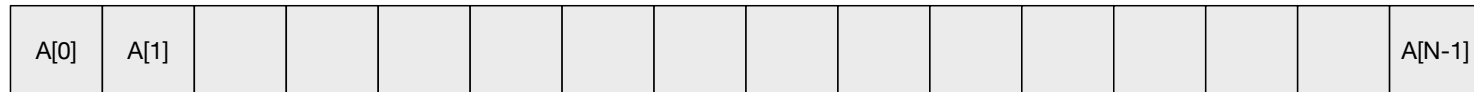
Reduction Operation

Reduction Operation

- A **reduction** operation reduces a set of values to a single value
 - Sum, Product, Minimum, Maximum are examples
- **Properties of reduction**
 - Associativity
 - Commutativity
 - Identity value
- Reduction is a key primitive for parallel computing
 - E.g., MapReduce programming model

Sequential Reduction

- A sequential implementation of reduction only needs a `for` loop to go through the whole input array
 - N elements \rightarrow N iterations



```
sum = 0; // Initialize with identity value
```

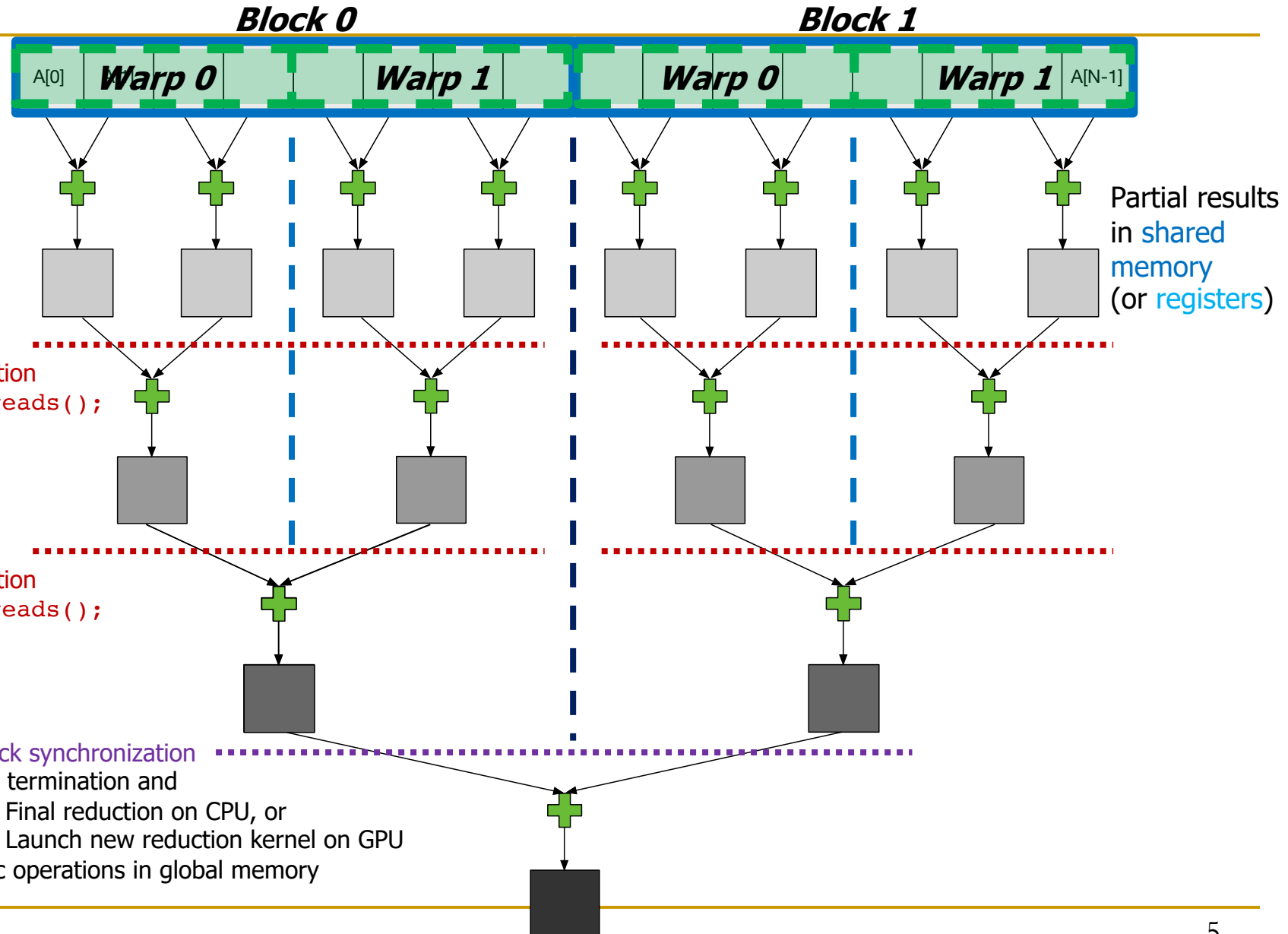
```
for(i = 0; i < N; ++i) {
```

```
    sum += A[i]; // Accumulate elements of input array A[]
```

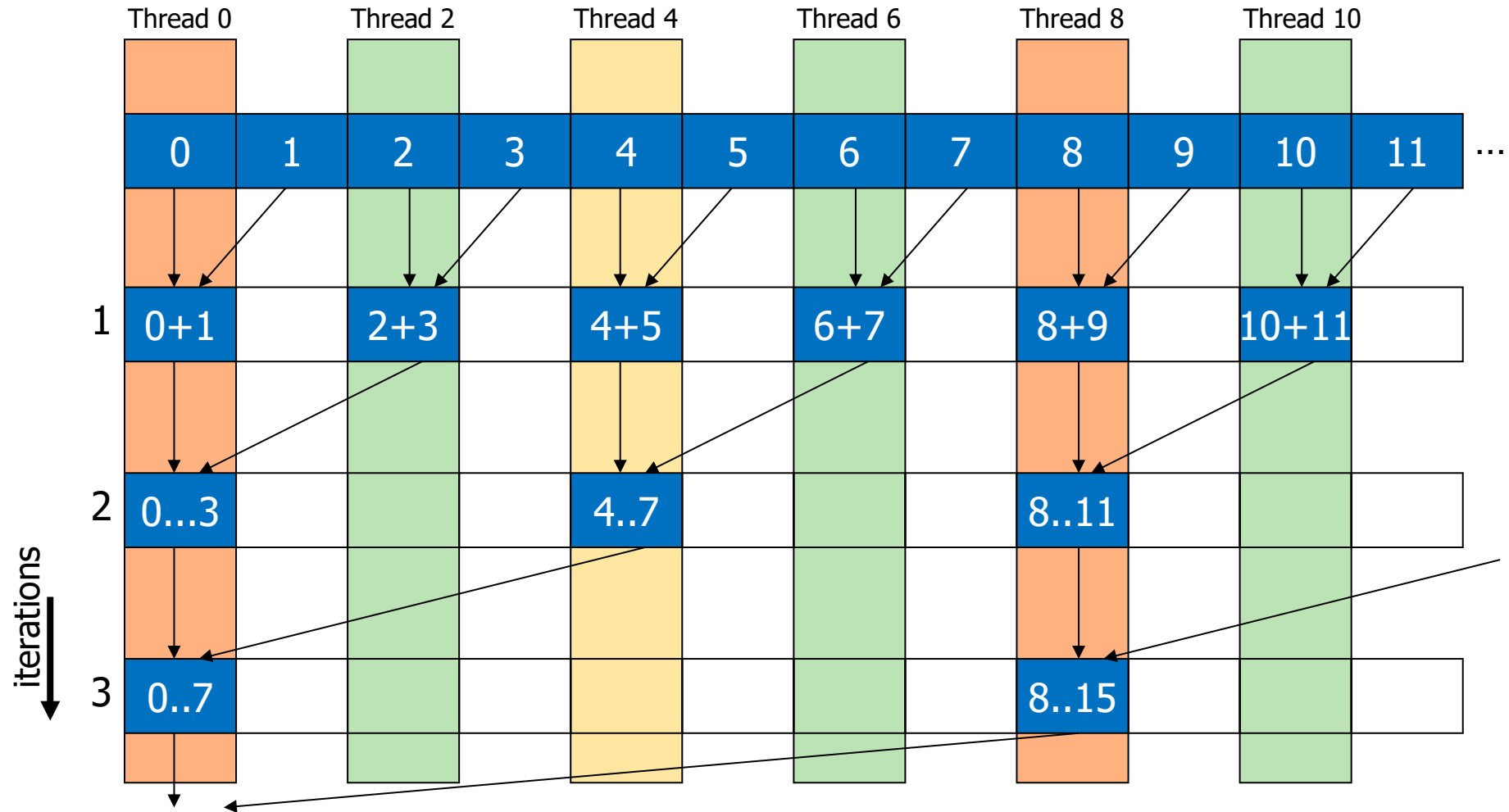
```
}
```

- Many independent operations
 - A parallel implementation can calculate multiple partial sums, and then reduce them

Tree-Based Reduction on GPU



Vector Reduction: Naïve Mapping (I)

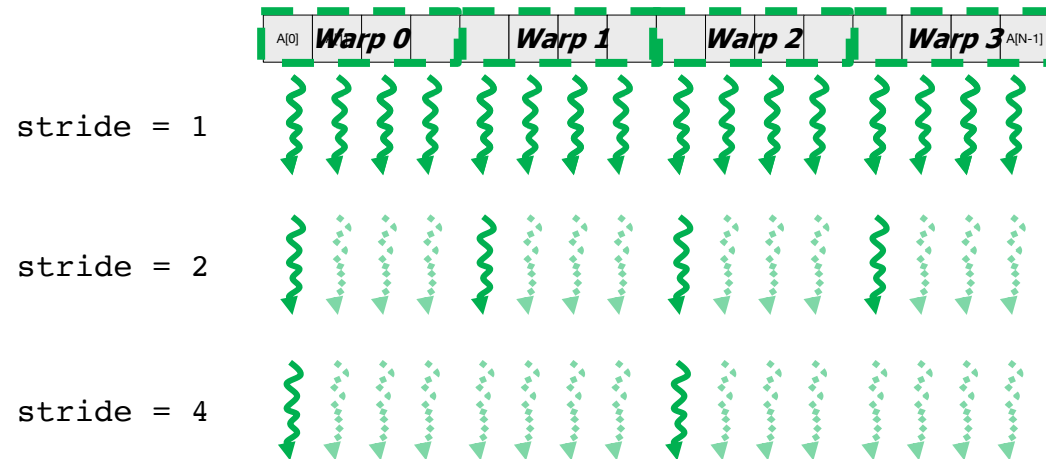


Vector Reduction: Naïve Mapping (II)

■ Program with **low SIMD utilization**

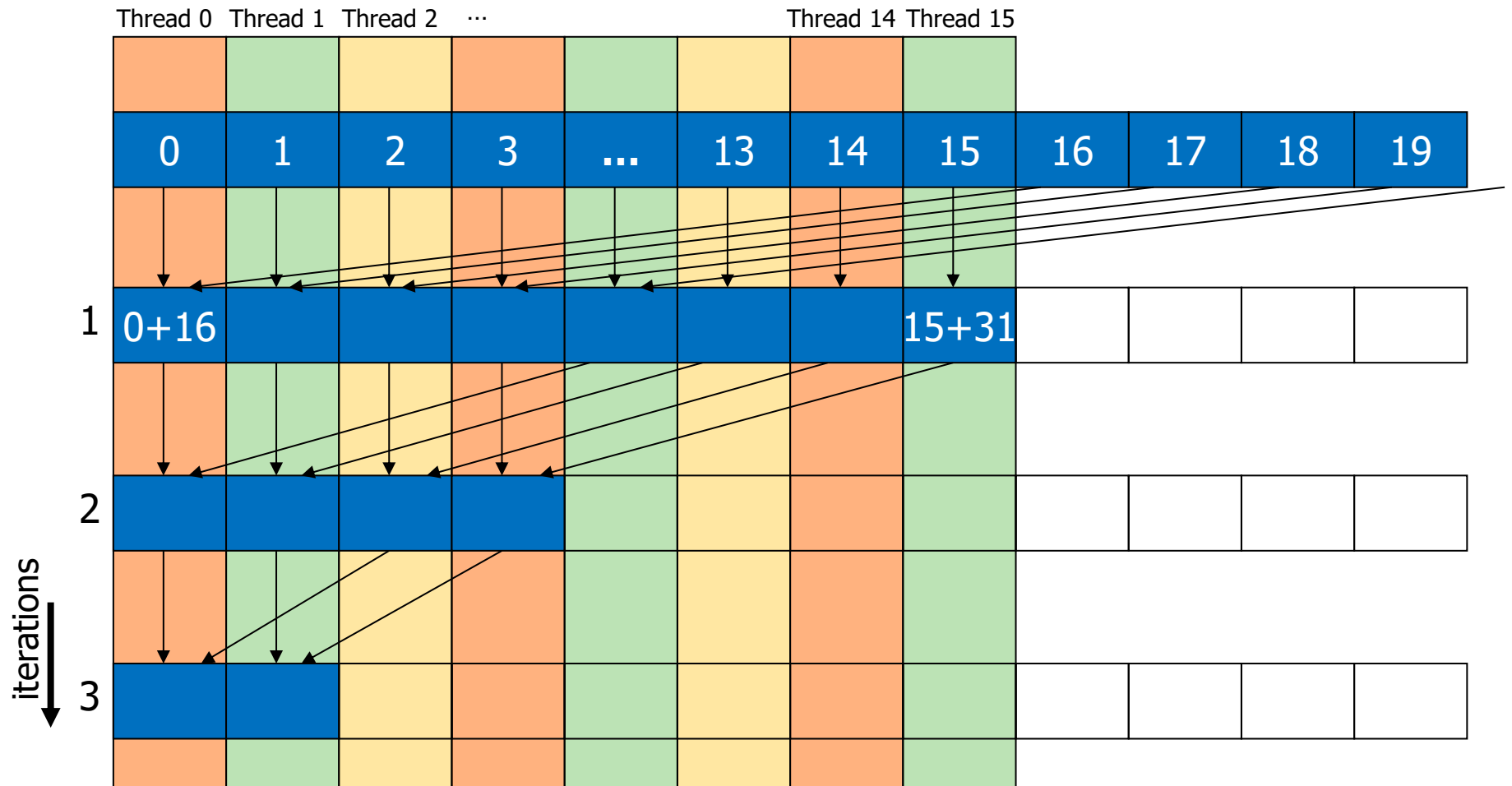
```
__shared__ float partialSum[]  
  
unsigned int t = threadIdx.x;  
  
for(int stride = 1; stride < blockDim.x; stride *= 2){  
  
    __syncthreads();  
  
    if (t % (2*stride) == 0)  
        partialSum[t] += partialSum[t + stride];  
  
}
```

How to avoid the
warp underutilization?



Divergence-Free Mapping (I)

- All active threads belong to the same warp

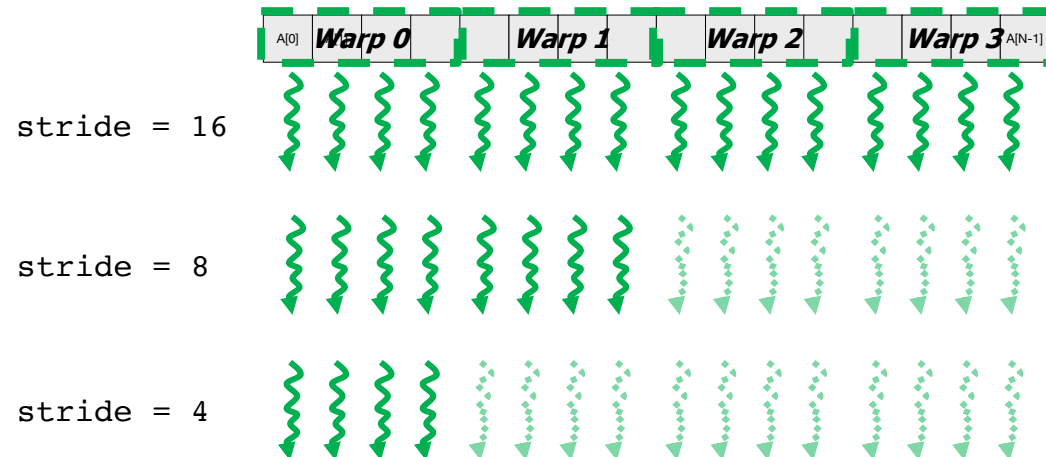


Divergence-Free Mapping (II)

■ Program with high SIMD utilization

```
__shared__ float partialSum[]  
  
unsigned int t = threadIdx.x;  
  
for(int stride = blockDim.x; stride > 0;  stride >> 1){  
  
    __syncthreads();  
  
    if (t < stride)  
        partialSum[t] += partialSum[t + stride];  
  
}
```

Warp utilization
is maximized



Warp Shuffle Functions

- Built-in **warp shuffle functions** enable threads to share data with other threads in the same warp
 - Faster than using shared memory and `__syncthreads()` to share across threads in the same block
- Variants:
 - `__shfl_sync(mask, var, srcLane)`
 - Direct copy from indexed lane
 - `__shfl_up_sync(mask, var, delta)`
 - Copy from a lane with lower ID relative to caller
 - `__shfl_down_sync(mask, var, delta)`
 - Copy from a lane with higher ID relative to caller
 - `__shfl_xor_sync(mask, var, laneMask)`
 - Copy from a lane based on bitwise XOR of own lane ID

Reduction with Warp Shuffle

```
__global__ void reduce_kernel(float* input, float* partialSums, unsigned int N) {

    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ float input_s[BLOCK_DIM];
    input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
    __syncthreads();

    // Reduction tree in shared memory
    for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) {
        if(threadIdx.x < stride) {
            input_s[threadIdx.x] += input_s[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Reduction tree with shuffle instructions
    float sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];

        for(unsigned int stride = WARP_SIZE/2; stride > 0; stride /= 2) {
            sum += __shfl_down_sync(0xffffffff, sum, stride);
        }
    }

    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = sum;
    }
}
```

Reduction with Warp Reduce

```
__global__ void reduce_kernel(int* input, int* partialSums, unsigned int N) {

    unsigned int segment = 2*blockDim.x*blockIdx.x;
    unsigned int i = segment + threadIdx.x;

    // Load data to shared memory
    __shared__ int input_s[BLOCK_DIM];
    input_s[threadIdx.x] = input[i] + input[i + BLOCK_DIM];
    __syncthreads();

    // Reduction tree in shared memory
    for(unsigned int stride = BLOCK_DIM/2; stride > WARP_SIZE; stride /= 2) {
        if(threadIdx.x < stride) {
            input_s[threadIdx.x] += input_s[threadIdx.x + stride];
        }
        __syncthreads();
    }

    // Reduction tree with shuffle instructions
    int sum;
    if(threadIdx.x < WARP_SIZE) {
        sum = input_s[threadIdx.x] + input_s[threadIdx.x + WARP_SIZE];

        // Warp reduce intrinsic for cc 8.0 or higher
        sum = __reduce_add_sync(0xffffffff, sum);
    }

    // Store partial sum
    if(threadIdx.x == 0) {
        partialSums[blockIdx.x] = sum;
    }
}
```

Atomic Operations

Atomic Operations (I)

- CUDA provides **atomic instructions** on shared memory and global memory
 - They perform **read-modify-write** operations atomically

- Arithmetic functions

- Add, sub, max, min, exch, inc, dec, CAS

`int atomicAdd(int*, int);`

Return value (old value)

Pointer to shared memory or global memory

Value to add

- Bitwise functions

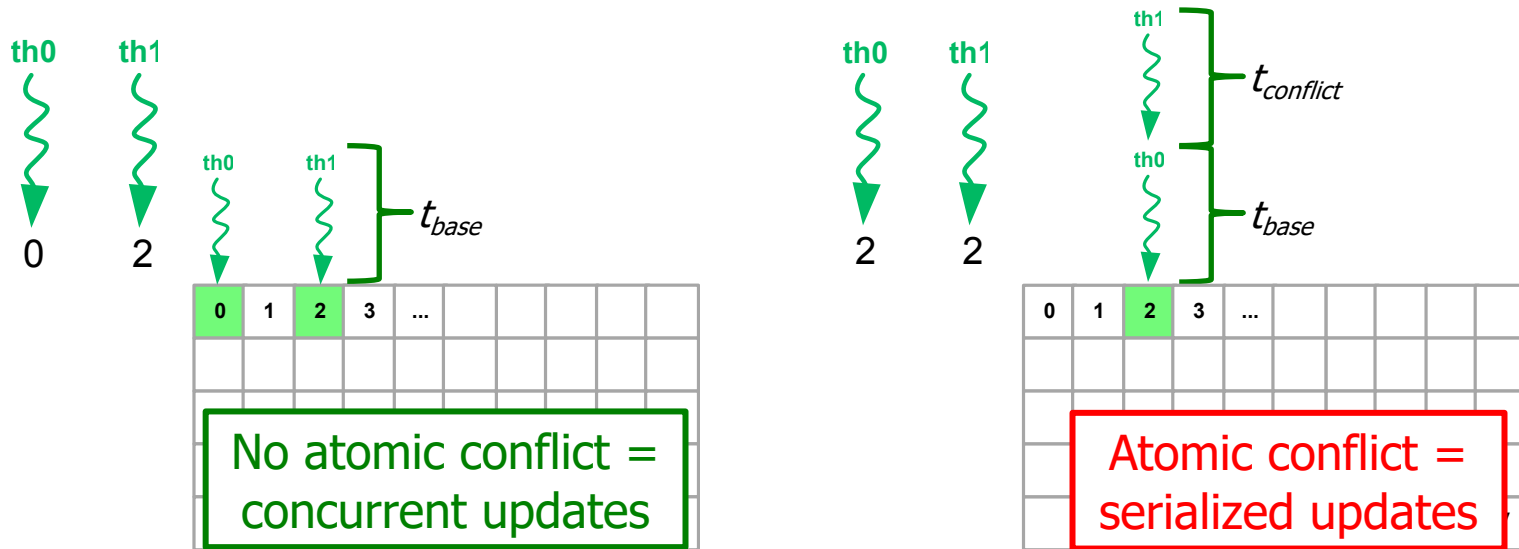
- And, or, xor

- Datatypes: int, uint, ull, float (half, single, double)*

* Datatypes for different atomic operations in <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#atomic-functions>

Atomic Operations (II)

- Atomic operations serialize the execution if there are atomic conflicts



Uses of Atomic Operations

■ Computation

- Atomics on an array that will be the output of the kernel
- Example
 - Histogram, reduction

■ Synchronization

- Atomics on memory locations that are used for synchronization or coordination
- Example
 - Counters, locks, flags...

- Use them to prevent **data races** when more than one thread need to update the same memory location

Data Races

- A **data race** occurs when multiple threads access the same memory location concurrently without ordering and at least one access is a write
 - Data races may result in unpredictable program output
- Example:

Thread A

```
oldVal = bins[b]  
newVal = oldVal + 1  
bins[b] = newVal
```

Thread B

```
oldVal = bins[b]  
newVal = oldVal + 1  
bins[b] = newVal
```

- If both threads have the same `b` and `bins[b]` is initially 0, the final value of `bins[b]` could be 2 or 1

Data Races Example (I)

Time	Thread A	Thread B
1	<code>oldVal = bins[b]</code>	
2	<code>newVal = oldVal + 1</code>	
3	<code>bins[b] = newVal</code>	
4		<code>oldVal = bins[b]</code>
5		<code>newVal = oldVal + 1</code>
6		<code>bins[b] = newVal</code>

In these two scenarios, the final value of `bins[b]` will be 2

Time	Thread A	Thread B
1		<code>oldVal = bins[b]</code>
2		<code>newVal = oldVal + 1</code>
3		<code>bins[b] = newVal</code>
4	<code>oldVal = bins[b]</code>	
5	<code>newVal = oldVal + 1</code>	
6	<code>bins[b] = newVal</code>	

Data Races Example (II)

Time	Thread A	Thread B
1	<code>oldVal = bins[b]</code>	
2	<code>newVal = oldVal + 1</code>	
3		<code>oldVal = bins[b]</code>
4	<code>bins[b] = newVal</code>	
5		<code>newVal = oldVal + 1</code>
6		<code>bins[b] = newVal</code>

In these two scenarios (and many others), the final value of bins[b] will be 1

Time	Thread A	Thread B
1		<code>oldVal = bins[b]</code>
2		<code>newVal = oldVal + 1</code>
3	<code>oldVal = bins[b]</code>	
4		<code>bins[b] = newVal</code>
5	<code>newVal = oldVal + 1</code>	
6	<code>bins[b] = newVal</code>	

Mutual Exclusion

- To avoid data races, concurrent read-modify-write operations to the same memory location need to be made **mutually exclusive** to enforce ordering
- One way to do this on CPUs is using **locks** (mutex)
 - Example:

```
mutex_lock(lock);  
  
++bins[b];  
  
mutex_unlock(lock);
```

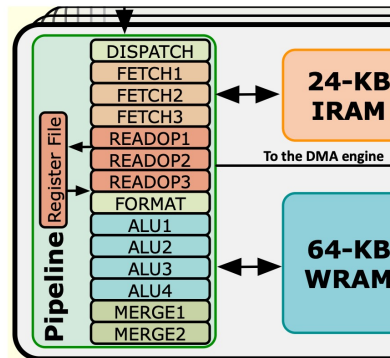
Using locks with SIMD execution may cause deadlock

Mutexes in a FGMT Architecture

- UPMEM Processing-in-Memory cores are fine-grained multithreaded
- Threads (called *tasklets*) can use mutexes for concurrent read-modify-write operations

DPU Pipeline

- In-order pipeline
 - Up to 425 MHz
- Fine-grain multithreaded
 - 24 hardware threads
- 14 pipeline stages
 - **DISPATCH**: Thread selection
 - **FETCH**: Instruction fetch
 - **READOP**: Register file
 - **FORMAT**: Operand formatting
 - **ALU**: Operation and WRAM
 - **MERGE**: Result formatting



SAFARI

31

Benchmarking a New Paradigm: An Experimental Analysis of a Real Processing-in-Memory Architecture

Juan Gómez-Luna¹ Izzat El Hajj² Ivan Fernandez^{1,3} Christina Giannoula^{1,4}
Geraldo F. Oliveira¹ Onur Mutlu¹

¹ETH Zürich ²American University of Beirut ³University of Malaga ⁴National Technical University of Athens

<https://arxiv.org/pdf/2105.03814.pdf>

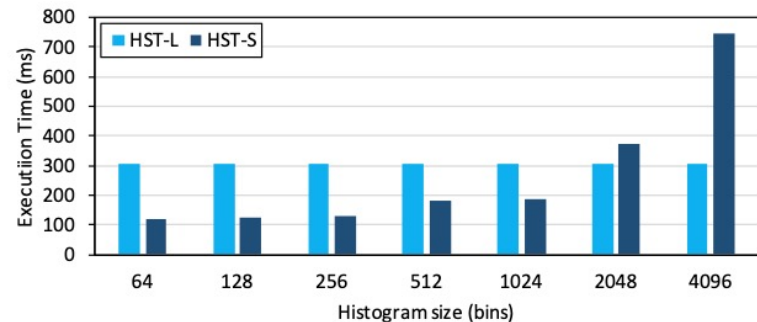


Figure 20: Execution times (ms) of two versions of histogram (HST-L, HST-S) on 1 DPU.

Atomic Operations: Architectural Support

- The GPU ISA evolves with GPU architecture generations
- CUDA: `int atomicAdd(int*, int);`
- PTX: `atom.shared.add.u32 %r25, [%rd14], %r24;`
- SASS:

Tesla, Fermi, Kepler

```
/*00a0*/ LDSLK P0, R9, [R8];  
/*00a8*/ @P0 IADD R10, R9, R7;  
/*00b0*/ @P0 STSCUL P1, [R8], R10;  
/*00b8*/ @!P1 BRA 0xa0;
```

Maxwell, Pascal, Volta...

```
/*01f8*/ ATOMS.ADD RZ, [R7], R11;
```

Native atomic operations for
32-bit integer, and 32-bit and
64-bit atomicCAS

Recall: Uses of Atomic Operations

■ Computation

- Atomics on an array that will be the output of the kernel
- Example
 - Histogram, reduction

■ Synchronization

- Atomics on memory locations that are used for synchronization or coordination
- Example
 - Counters, locks, flags...

- Use them to prevent **data races** when more than one thread need to update the same memory location

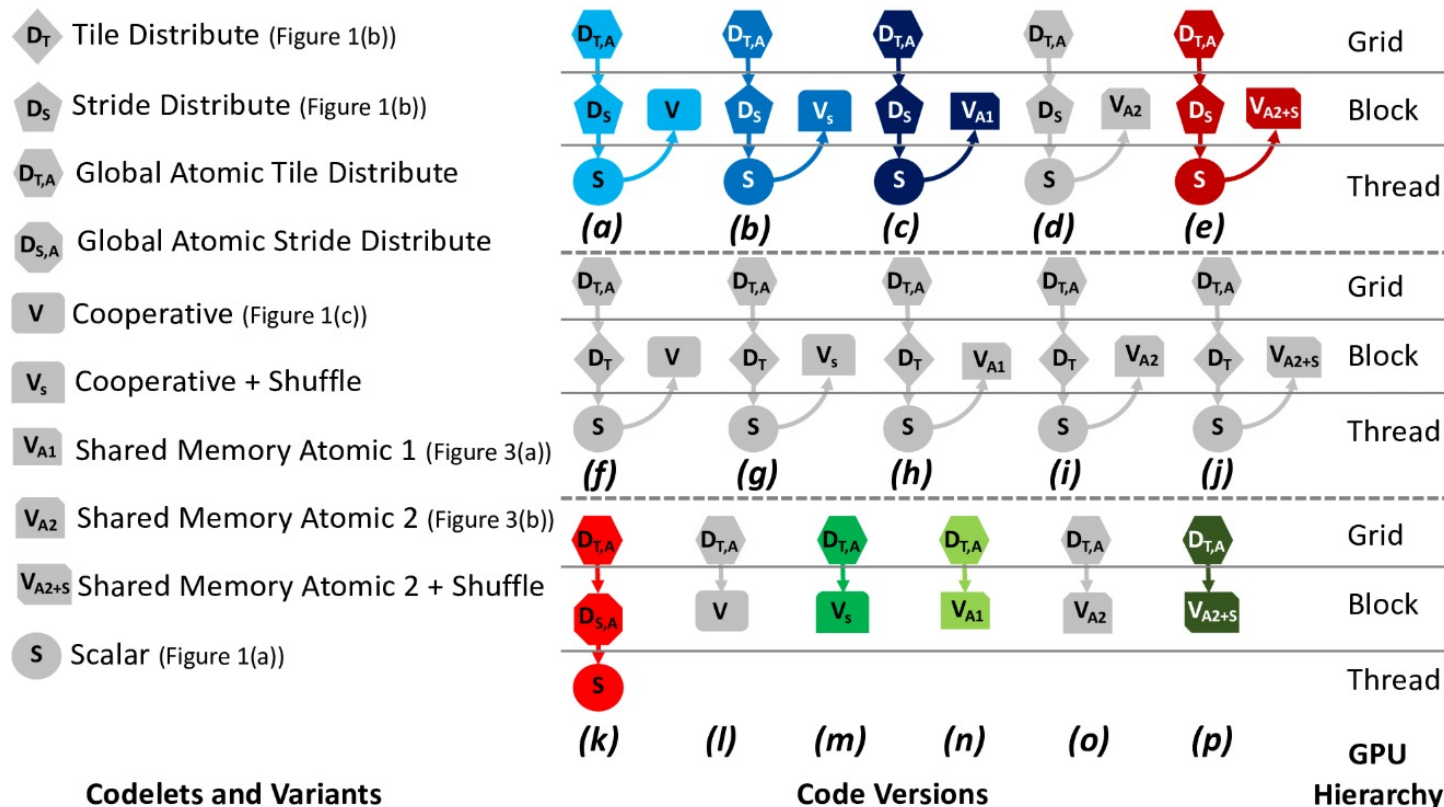
Optimized Parallel Reduction

- 7 versions in CUDA samples: Tree-based reduction in shared memory
 - ❑ Version 0: No whole warps active
 - ❑ Version 1: Contiguous threads, but many bank conflicts
 - ❑ Version 2: No bank conflicts
 - ❑ Version 3: First level of reduction when reading from global memory
 - ❑ Version 4: Warp shuffle or unrolling of final warp
 - ❑ Version 5: Warp shuffle or complete unrolling
 - ❑ Version 6: Multiple elements per thread sequentially

Reduction with Atomic Operations

- 3 new versions of reduction based on 3 previous versions
 - Version 0: No whole warps active
 - Version 3: First level of reduction when reading from global memory
 - Version 6: Multiple elements per thread sequentially
- New versions 7, 8, and 9
 - Replace the `for` loop (tree-based reduction) with one shared memory atomic operation per thread

Search Space of Parallel Reduction



Over 85 different versions possible!

Automatic Generation of Parallel Reduction

- Simon Garcia De Gonzalo, Sitao Huang, Juan Gomez-Luna, Simon Hammond, Onur Mutlu, and Wen-mei Hwu,
"Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs"
Proceedings of the International Symposium on Code Generation and Optimization (CGO), Washington, DC, USA, February 2019.
[[Slides \(pptx\)](#) ([pdf](#))]

Automatic Generation of Warp-Level Primitives and Atomic Instructions for Fast and Portable Parallel Reduction on GPUs

Simon Garcia De Gonzalo
CS and Coordinated Science Lab
UIUC
grcdgnz2@illinois.edu

Sitao Huang
ECE and Coordinated Science Lab
UIUC
shuang91@illinois.edu

Juan Gómez-Luna
Computer Science
ETH Zurich
juang@ethz.ch

Simon Hammond
Scalable Computer Architecture
Sandia National Laboratories
sdhammo@sandia.gov

Onur Mutlu
Computer Science
ETH Zurich
omutlu@ethz.ch

Wen-mei Hwu
ECE and Coordinated Science Lab
UIUC
w-hwu@illinois.edu

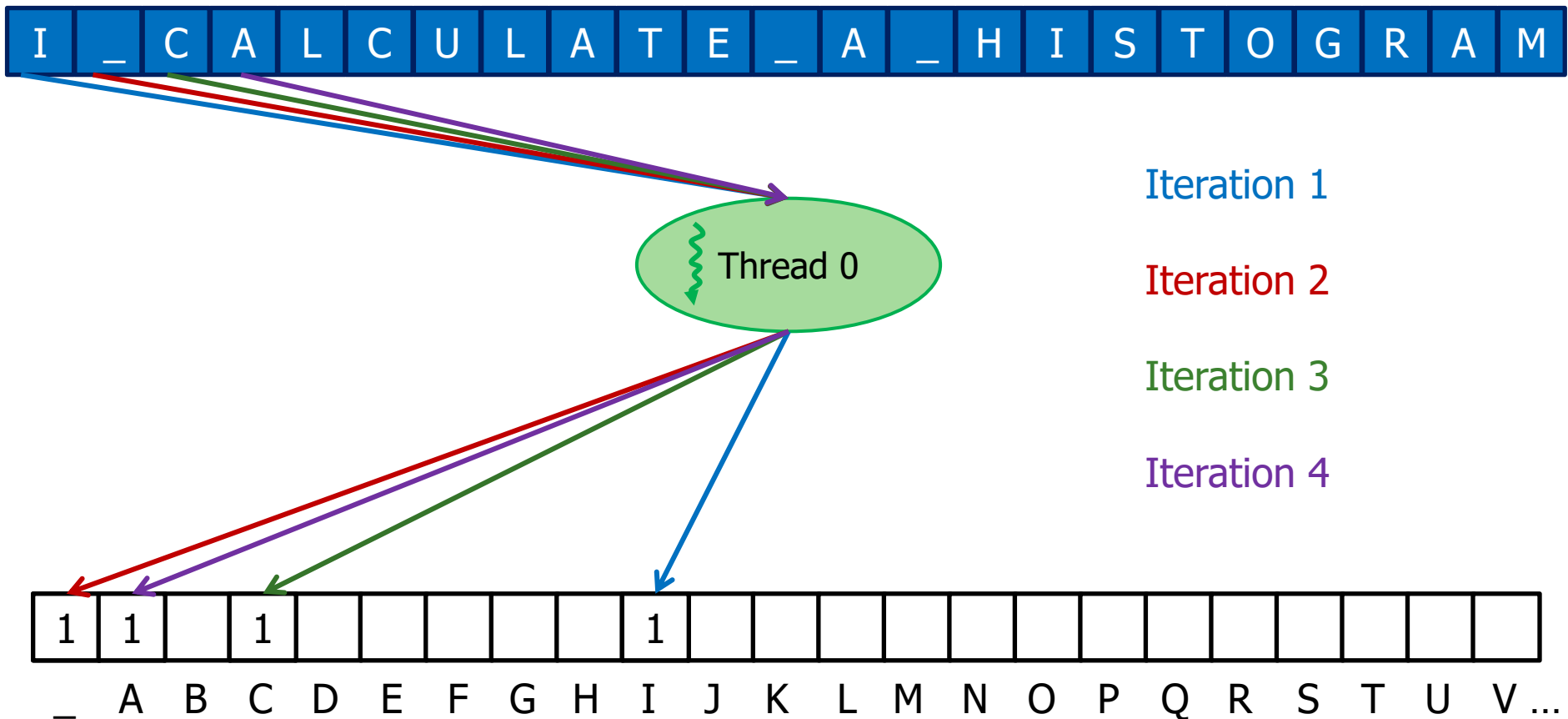
Histogram Computation

Histogram Computation

- Histogram is a frequently used computation for **reducing the dimensionality and extracting notable features** and patterns from large data sets
 - Feature extraction for object recognition in images
 - Fraud detection in credit card transactions
 - Correlating heavenly object movements in astrophysics
 - ...
- Basic histograms - for **each element in the data set, use the value to identify a "bin" to increment**
 - Divide possible input value range into "bins"
 - Associate a counter to each bin
 - For each input element, examine its value and determine the bin it falls into and increment the counter for that bin

Sequential Histogram Computation

- A sequential implementation of histogram computation reads all input elements one by one and updates the corresponding histogram bins

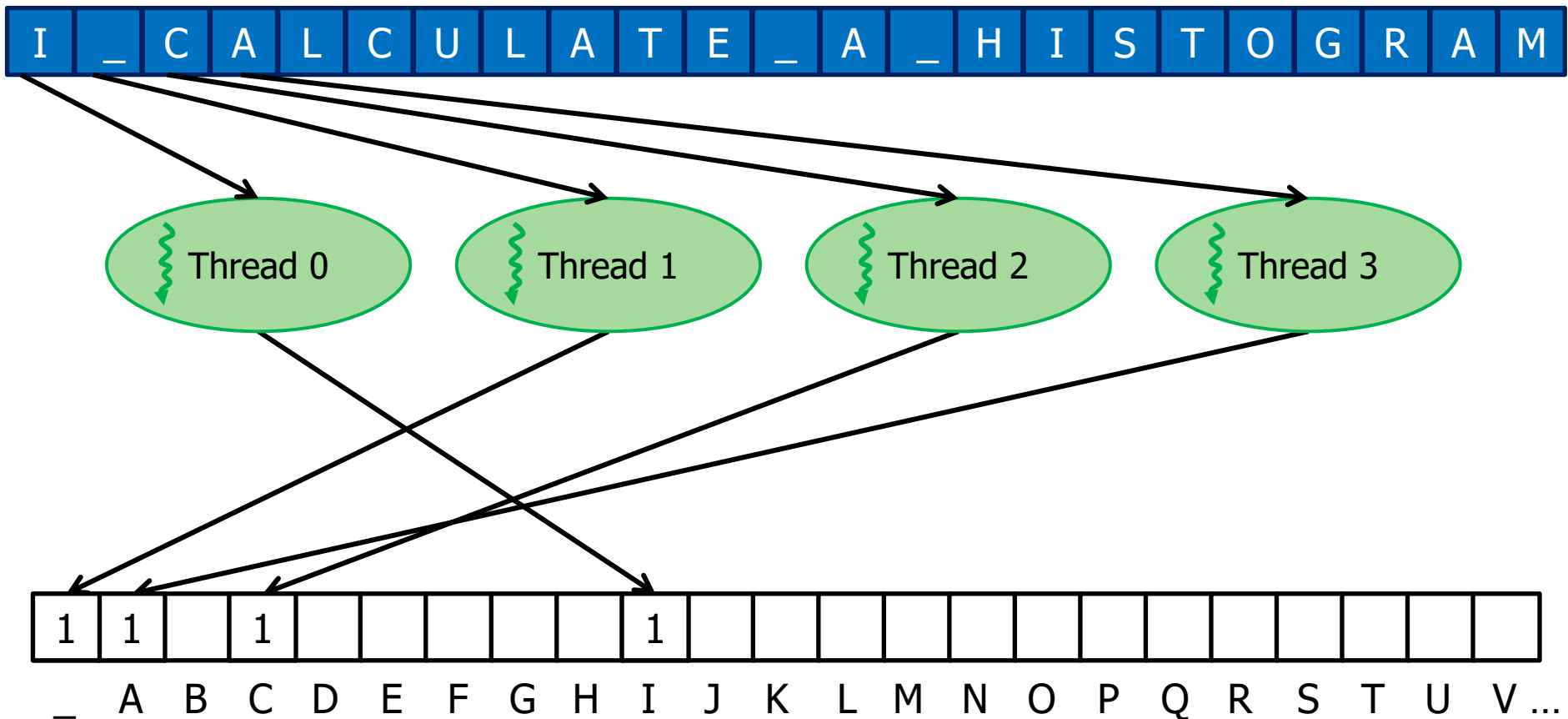


Sequential Histogram Function

```
void histogram_calculation(unsigned int *histo,  
                           unsigned int *input,  
                           unsigned int input_size){  
  
    int i = 0; // Loop index  
  
    while(i < input_size){  
        unsigned int val = input[i];  
  
        histo[val] += 1;  
  
        i++;  
    }  
}
```

Parallel Histogram Computation: Iteration 1

- Adjacent threads read adjacent input characters
 - Reads from the input array are **coalesced**



(Wrong) Parallel Histogram Kernel

```
__global__ void histogram_kernel(unsigned int *histo,
                                unsigned int *input,
                                unsigned int input_size){

    int i = blockIdx.x * blockDim.x + threadIdx.x; // Thread index

    int stride = blockDim.x * gridDim.x; // Total number of threads

    while(i < input_size){

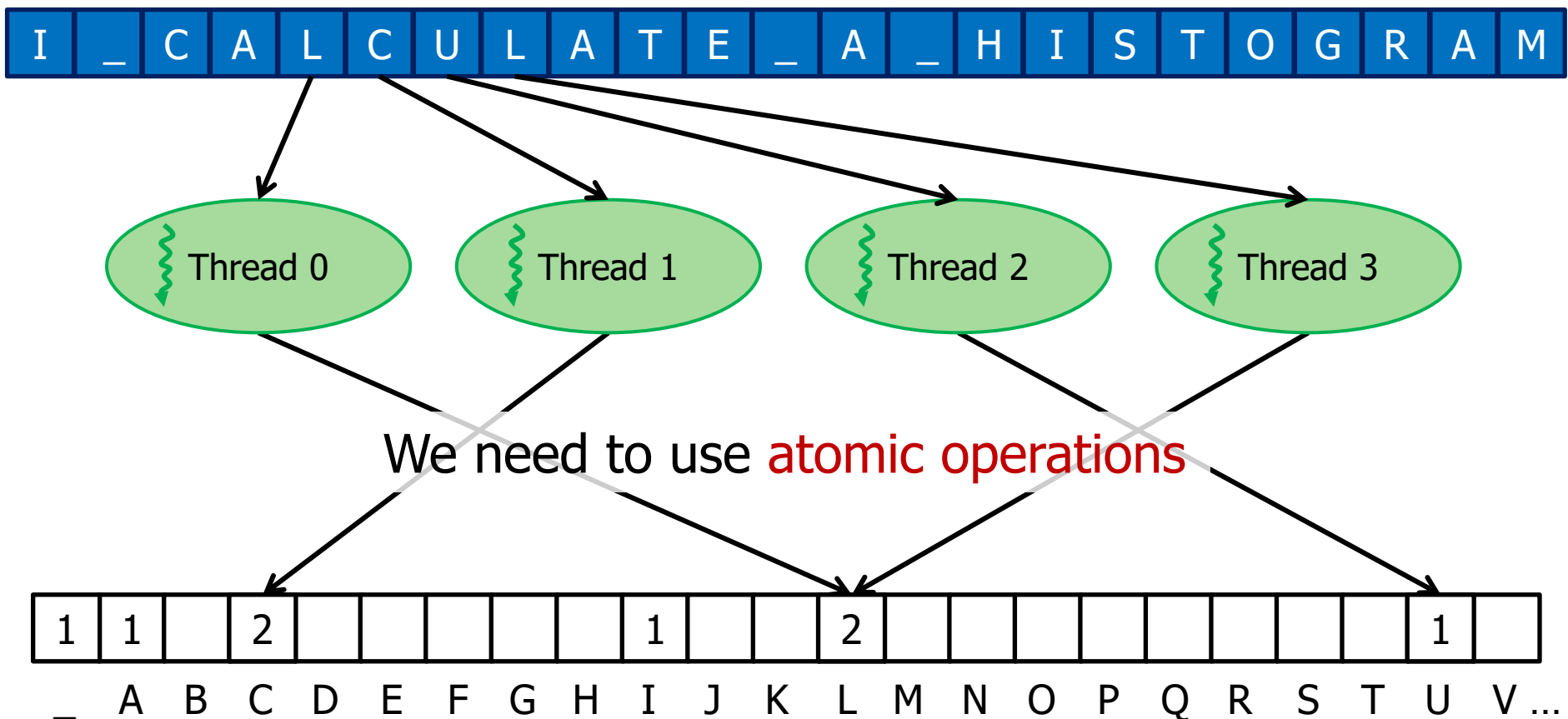
        unsigned int val = input[i];

        histo[val] += 1;

        i += stride;
    }
}
```

Parallel Histogram Computation: Iteration 2

- All threads move to the next section of the input
 - Each thread moves to element $\text{threadID} + \#\text{threads}$



(Correct) Parallel Histogram Kernel

```
__global__ void histogram_kernel(unsigned int *histo,
                                unsigned int *input,
                                unsigned int input_size){

    int i = blockIdx.x * blockDim.x + threadIdx.x; // Thread index

    int stride = blockDim.x * gridDim.x; // Total number of threads

    while(i < input_size){

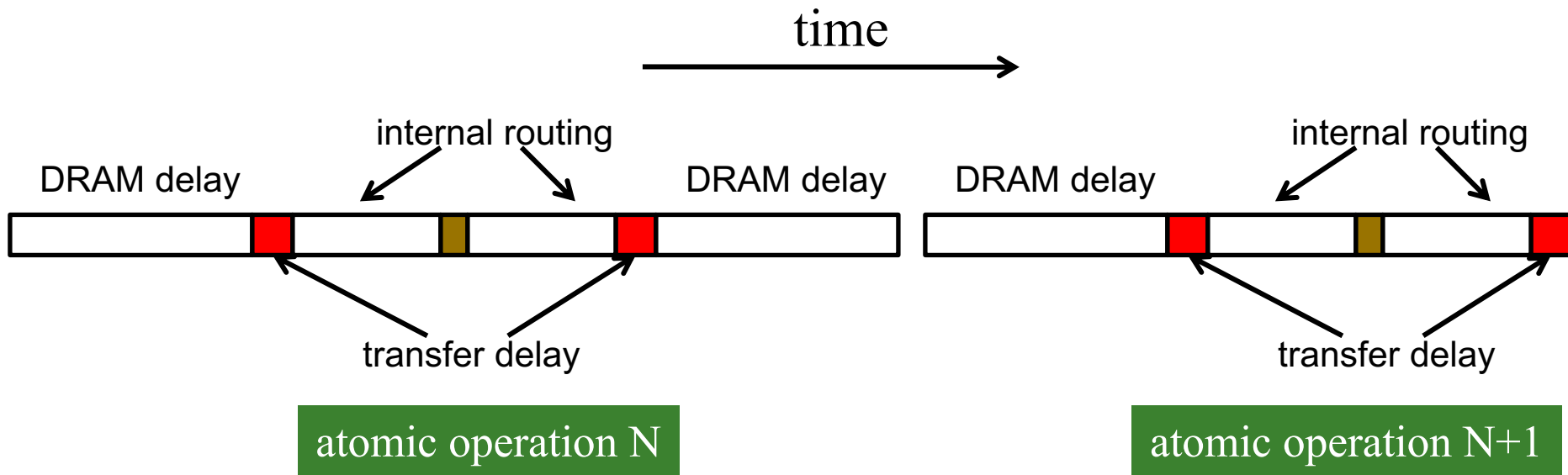
        unsigned int val = input[i];

        atomicAdd(&histo[val], 1);

        i += stride;
    }
}
```

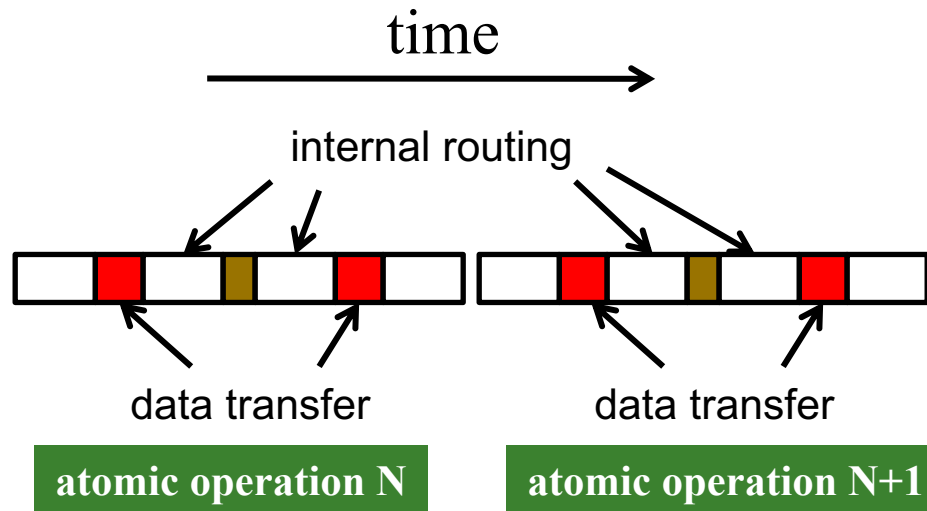
Atomic Operations on DRAM

- Each Load-Modify-Store has two full memory access delays
 - All atomic operations on the same variable (RAM location) are serialized



Hardware Improvements

- Atomic operations on Fermi L2 cache
 - ❑ Medium latency, but still serialized
 - ❑ Global to all blocks
 - ❑ “Free improvement” on Global Memory atomics



Hardware Improvements (Cont.)

- Atomic operations on Shared Memory
 - Very short latency, but still serialized
 - Private to each thread block
 - Need algorithm work by programmers (more later)

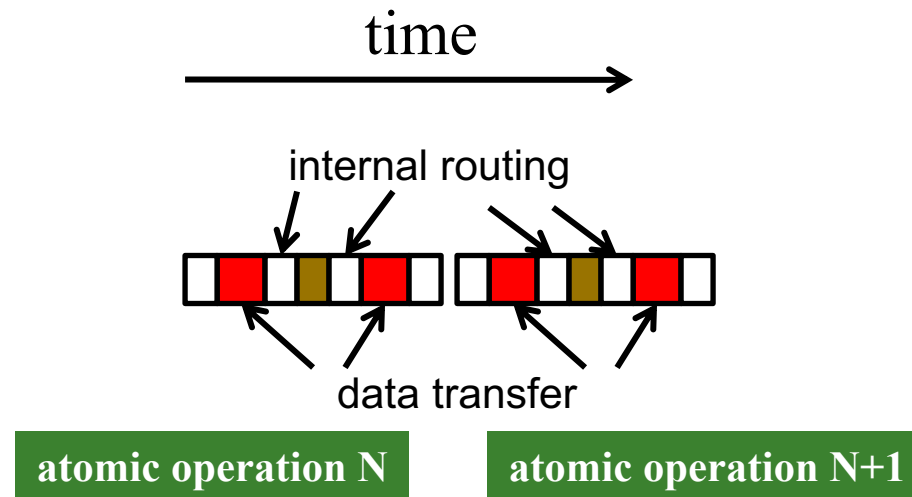
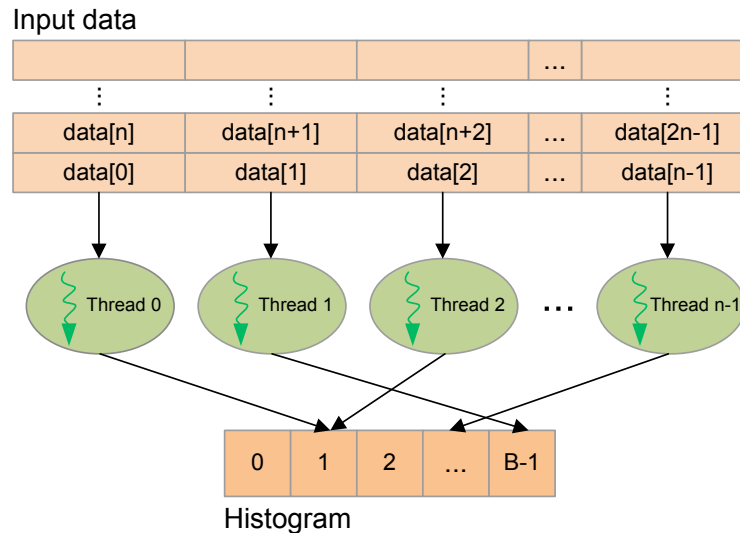


Image Histogram

- Histograms are widely used in **image processing**
 - Some **computation before voting** in the histogram may be needed

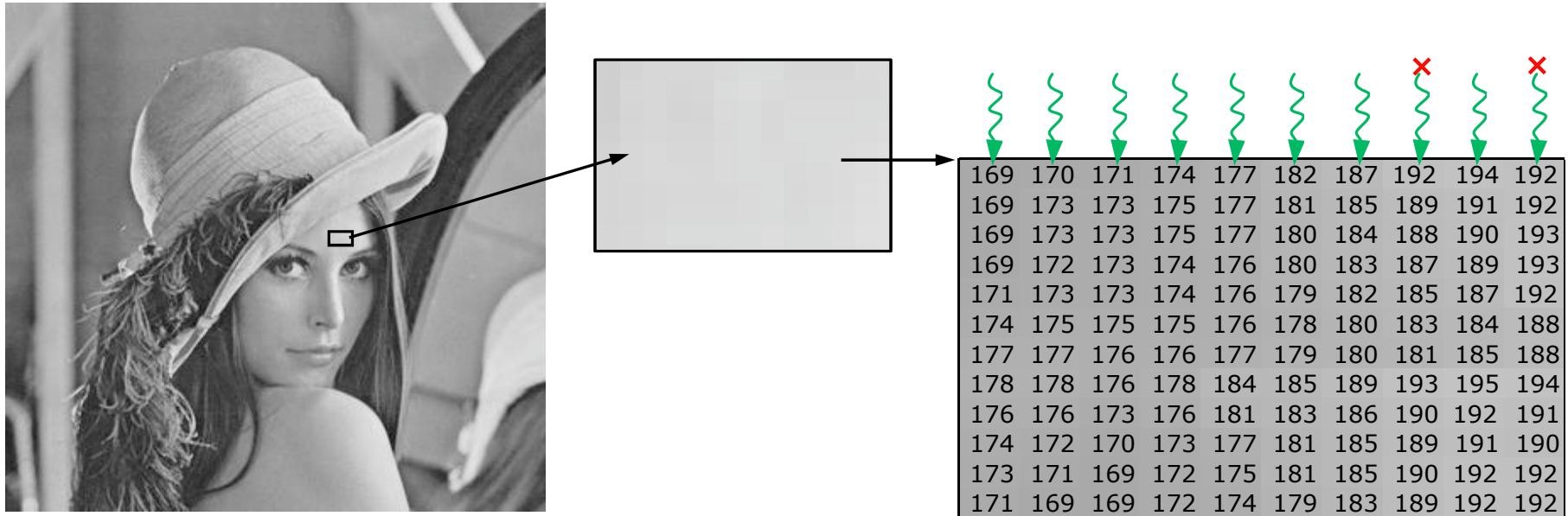
```
For (each pixel i in image I){  
    Pixel = I[i]                // Read pixel  
    Pixel' = Computation(Pixel) // Optional computation  
    Histogram[Pixel']++         // Vote in histogram bin  
}
```

- Parallel threads frequently incur **atomic conflicts** in image histogram computation



Histogram Computation of Natural Images

- **Frequent atomic conflicts** due to the spatial similarity of the pixel value distribution in natural images



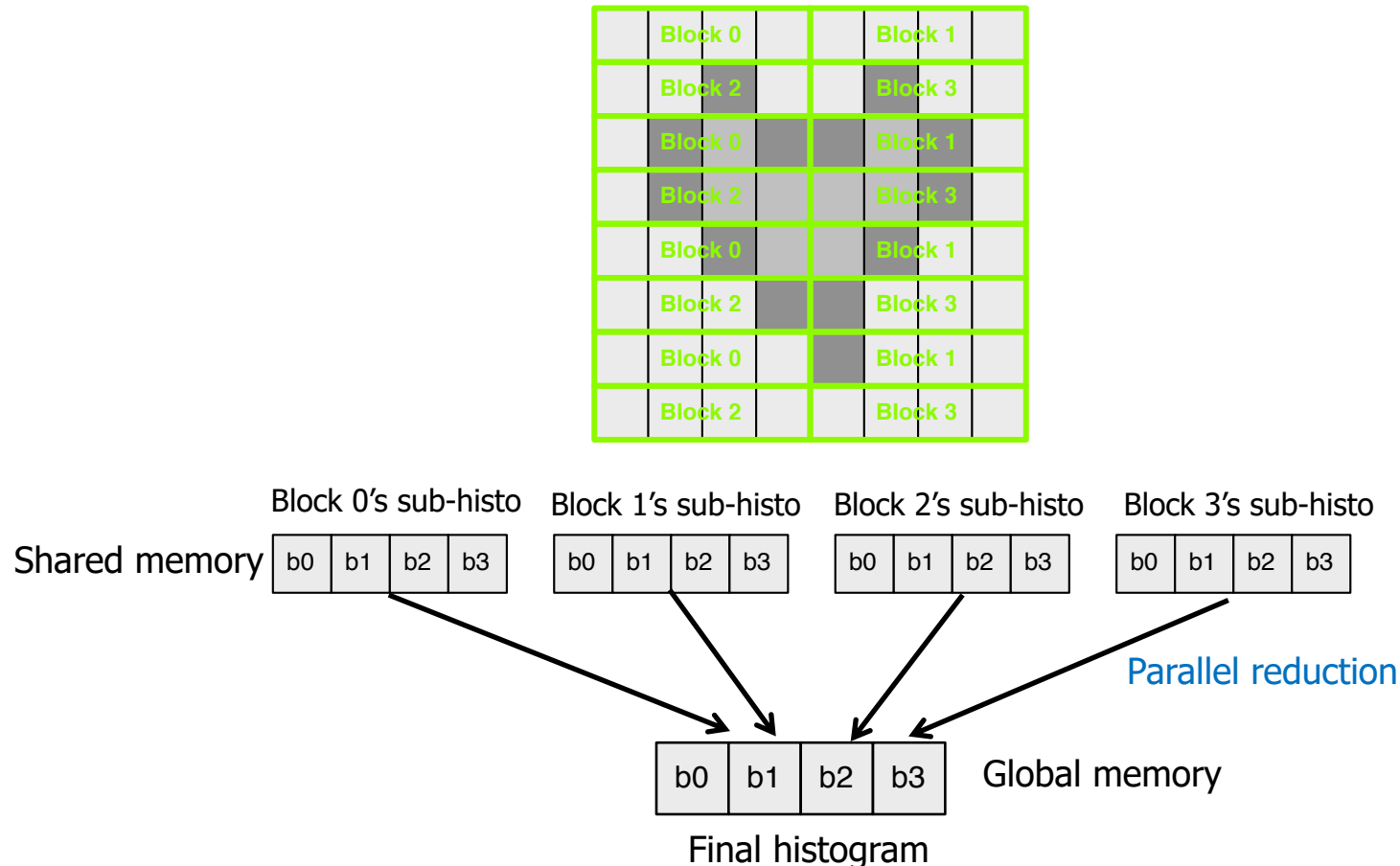
- By using **multiple sub-histograms** (which are merged at the end), we can reduce the frequency of atomic conflicts
- This optimization technique is called **privatization**

Privatization

- **Privatization** is an optimization technique where multiple private copies of an output are maintained, then the global copy is updated on completion
 - Operations on the output must be **associative and commutative** because the order of updates has changed
- **Advantages:**
 - Reduces contention on the global copy
 - If the output is small enough, the **private copy can be placed in shared memory** reducing access latency

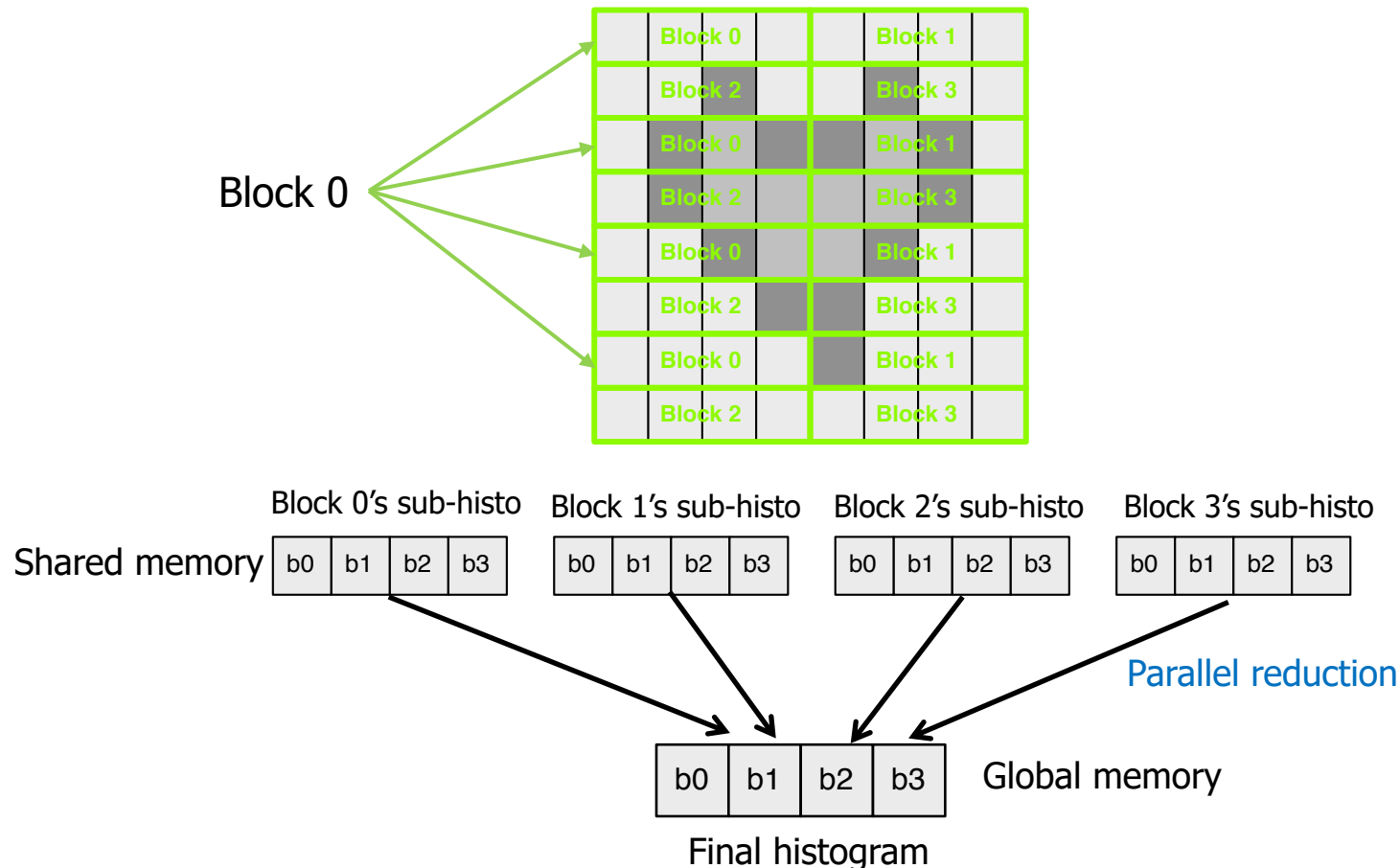
Histogram Privatization

- **Privatization:** Per-block sub-histograms in shared memory
 - Threads use atomic operations in shared memory



Histogram Privatization + Coarsening

- **Coarsening**: Each block processes several image chunks
 - Fewer sub-histograms to initialize and to merge at the end



Parallel Histogram Kernel with Privatization

(+ Coarsening)

```
__global__ void histogram_kernel(unsigned int *histo, unsigned int *input, unsigned int input_size){

    int tid = blockIdx.x * blockDim.x + threadIdx.x; // Thread index
    int stride = blockDim.x * gridDim.x; // Total number of threads

    __shared__ unsigned int histo_s[BINS]; // Private per-block sub-histogram

    // Sub-histogram initialization
    for(int i = threadIdx.x; i < BINS; i += blockDim.x) {
        histo_s[i] = 0;
    }
    __syncthreads(); // Intra-block synchronization

    // Main loop to compute per-block sub-histograms
    for(int i = tid; i < input_size ; i += stride) {

        unsigned int val = input[i]; // Global memory read (coalesced)

        atomicAdd(&histo_s[val], 1); // Atomic addition in shared memory
    }
    __syncthreads(); // Intra-block synchronization

    // Merge per-block sub-histograms and write to global memory
    for(int i = threadIdx.x; i < BINS; i += blockDim.x) {

        // Atomic addition in global memory
        atomicAdd(histo + i, histo_s[i]);
    }
}
```

Warp-Synchronous Programming for Atomic Operations

Warp Shuffle Functions

- Built-in **warp shuffle functions** enable threads to share data with other threads in the same warp
 - Faster than using shared memory and `__syncthreads()` to share across threads in the same block
- Variants:
 - `__shfl_sync(mask, var, srcLane)`
 - Direct copy from indexed lane
 - `__shfl_up_sync(mask, var, delta)`
 - Copy from a lane with lower ID relative to caller
 - `__shfl_down_sync(mask, var, delta)`
 - Copy from a lane with higher ID relative to caller
 - `__shfl_xor_sync(mask, var, laneMask)`
 - Copy from a lane based on bitwise XOR of own lane ID

Other Warp-Synchronous Primitives

- `__syncwarp(unsigned)`

Forces the reconvergence of the threads in the mask

- `__activemask()`

Returns the mask of converged threads

- `__all_sync(unsigned, bool)` and
`__any_sync(unsigned, bool)`

Returns true if all or any of the participating threads pass true

Other Warp-Synchronous Primitives

- `__ballot_sync(unsigned, bool)`

Returns the mask of threads that passed true

- `__match_all_sync(unsigned, _T)`

Returns true if all participating threads pass the same value

- `__match_any_sync(unsigned, _T)`

Returns the mask of participating threads passing the same value

Coalesced Atomic Operations

- Identify threads operating on the same atomic and use a reduction

```
int atomic_add(int * ptr, int value){  
  
    unsigned active_mask = __activemask();  
    unsigned active_mask = __match_any_sync(active_mask, ptr);  
  
    int value = reduce_warp(active_mask, value);  
  
    if((__ffs(active_mask) - 1) == lane) {  
        value = atomicAdd(ptr, value);  
    }  
  
    value = __shfl_sync(active_mask, value, __ffs(active_mask) - 1);  
    return value;  
}
```

Evolution of the Architectural Support for Atomic Operations

Atomic Operations on Shared Memory

- The architectural support for atomic operations evolves across GPU generations
- CUDA: `int atomicAdd(int*, int);`
- PTX: `atom.shared.add.u32 %r25, [%rd14], %r24;`
- SASS:

Tesla, Fermi, Kepler

```
/*00a0*/ LDSLK P0, R9, [R8];  
/*00a8*/ @P0 IADD R10, R9, R7;  
/*00b0*/ @P0 STSCUL P1, [R8], R10;  
/*00b8*/ @!P1 BRA 0xa0;
```

Maxwell, Pascal, Volta...

```
/*01f8*/ ATOMS.ADD RZ, [R7], R11;
```

Native atomic operations for
32-bit integer, and 32-bit and
64-bit atomicCAS

Lock-Free Mechanism for Shared Memory Atomics

- Tesla, Fermi, and Kepler architectures
 - Parallel processing unit (PPU) = GPU core

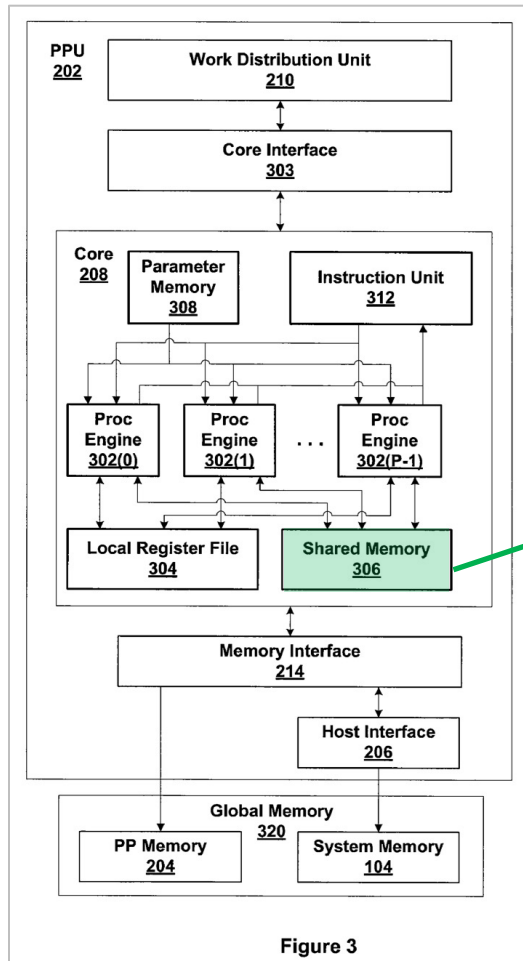


Figure 3

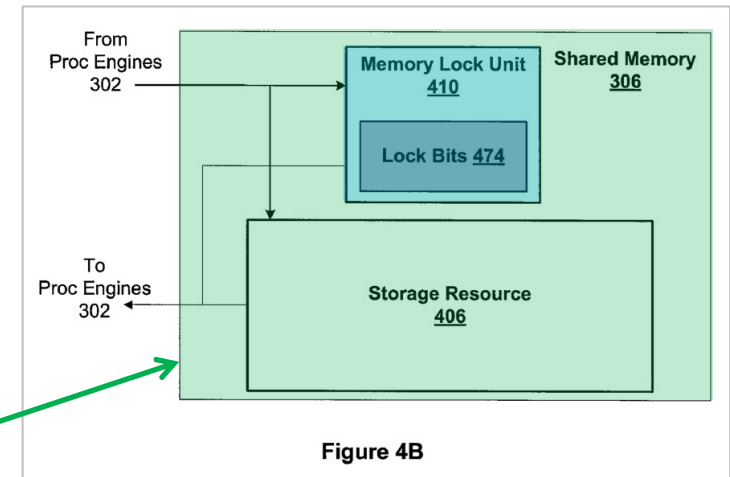
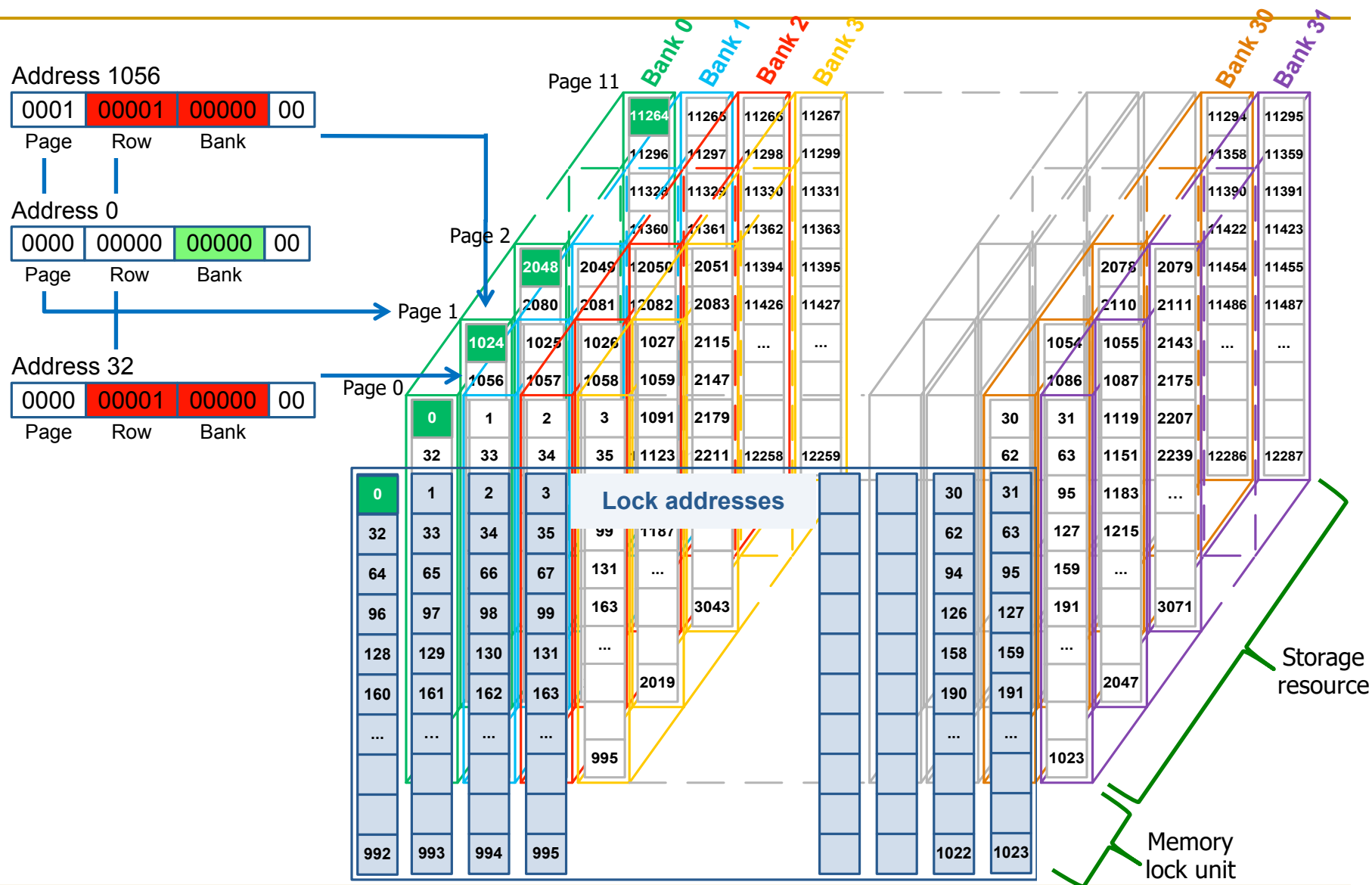


Figure 4B

```
/*00a0*/ LDSLK P0, R9, [R8];  
/*00a8*/ @P0 IADD R10, R9, R7;  
/*00b0*/ @P0 STSCUL P1, [R8], R10;  
/*00b8*/ @!P1 BRA 0xa0;
```

Shared Memory Organization



Assembly Code for Shared Memory Atomics

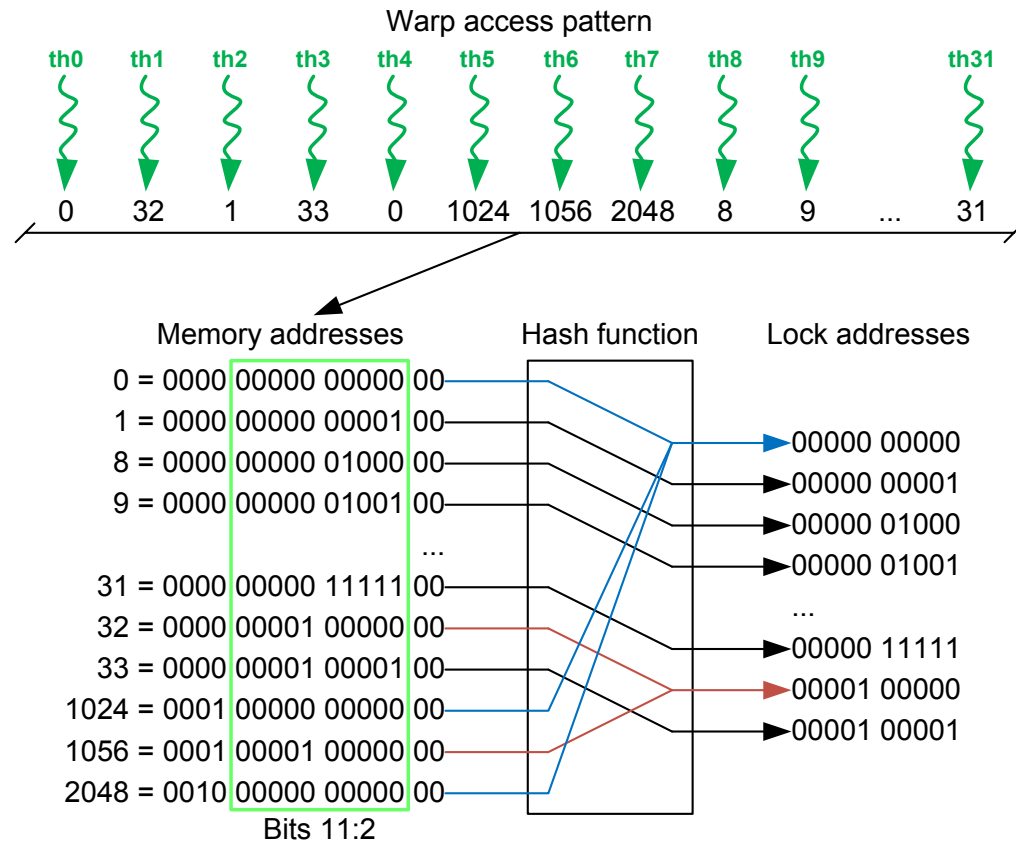
(pre Maxwell)

- Lock-free mechanism
 - Predicated execution
- **LDSLK** loads from shared memory and sets one lock bit
 - Predication register P0 set if lock succeeds
- **STSCUL** stores and releases the lock
- **BRA** jumps to start a new attempt to acquire the lock

```
/*0090*/      PSETP.AND.AND P1, PT, !PT, PT, PT; // Predicate set predicate
/*0098*/      SSY 0xd0;                          // Set synchronization point
/*00a0*/      LDSLK P0, R9, [R8];                 // Load and lock
/*00a8*/      @P0 IADD R10, R9, R7;              // Integer addition
/*00b0*/      @P0 STSCUL P1, [R8], R10;          // Store conditionally and unlock
/*00b8*/      @!P1 BRA 0xa0;                     // Predicated unconditional branch
/*00c8*/      Instruction.S
/*00d0*/      ...
```

Limited Number of Lock Bits

- The limited number of locks may cause **high contention**
 - 256 lock bits in Tesla, 1024 lock bits in Fermi and Kepler

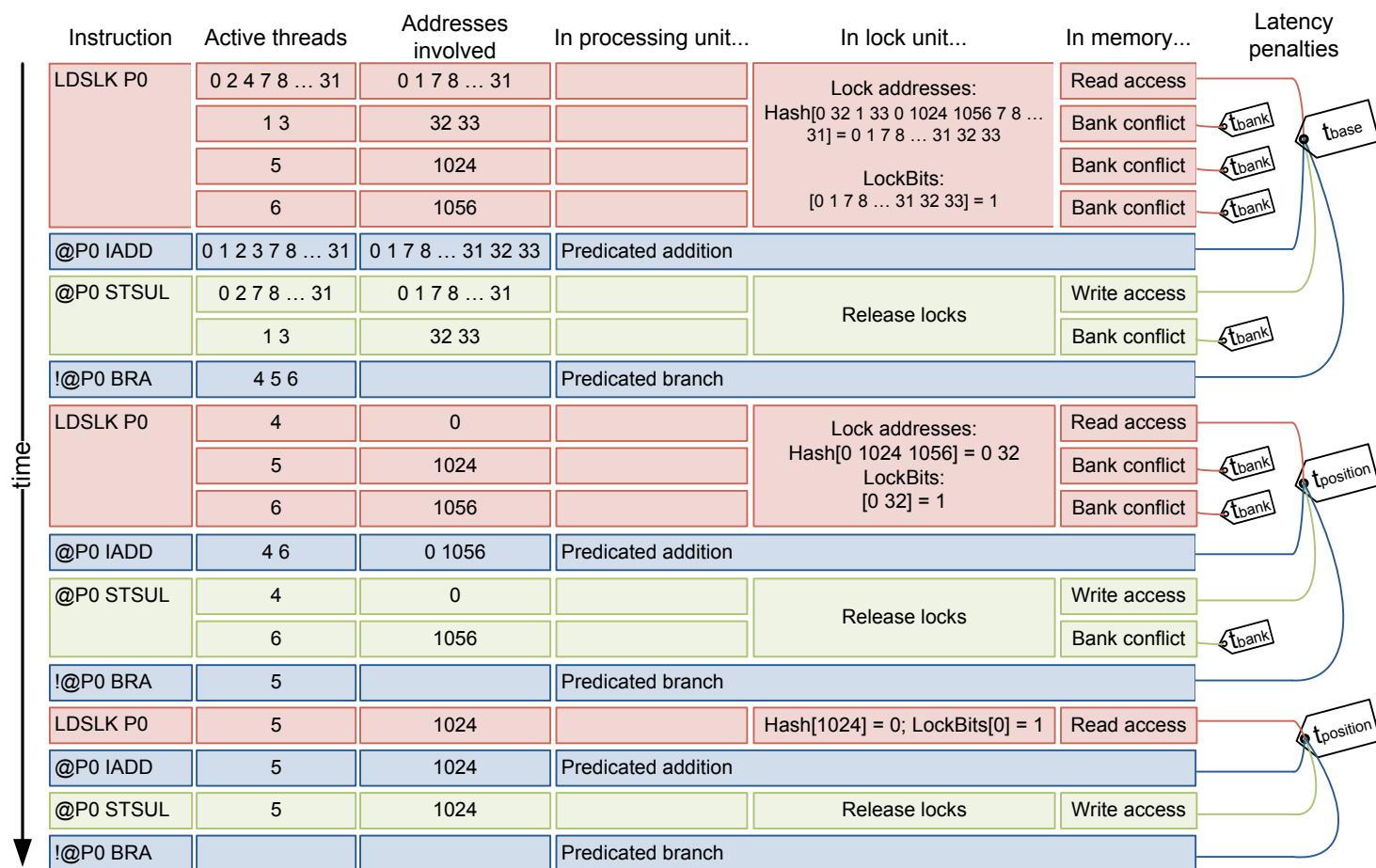
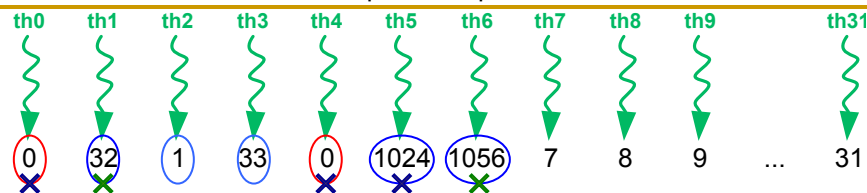


Example Execution Timeline

```

/*0210*/ LDSLK P0, R7, [R9];
/*0218*/ @P0 IADD R10, R7, 0x1;
/*0220*/ @P0 STSUL [R9], R10;
/*0228*/ @!P0 BRA 0x210;
    
```

Warp access pattern



Microbenchmarking Atomic Operations

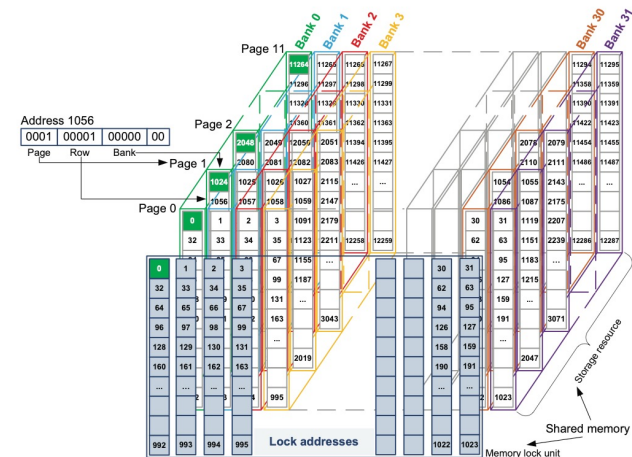
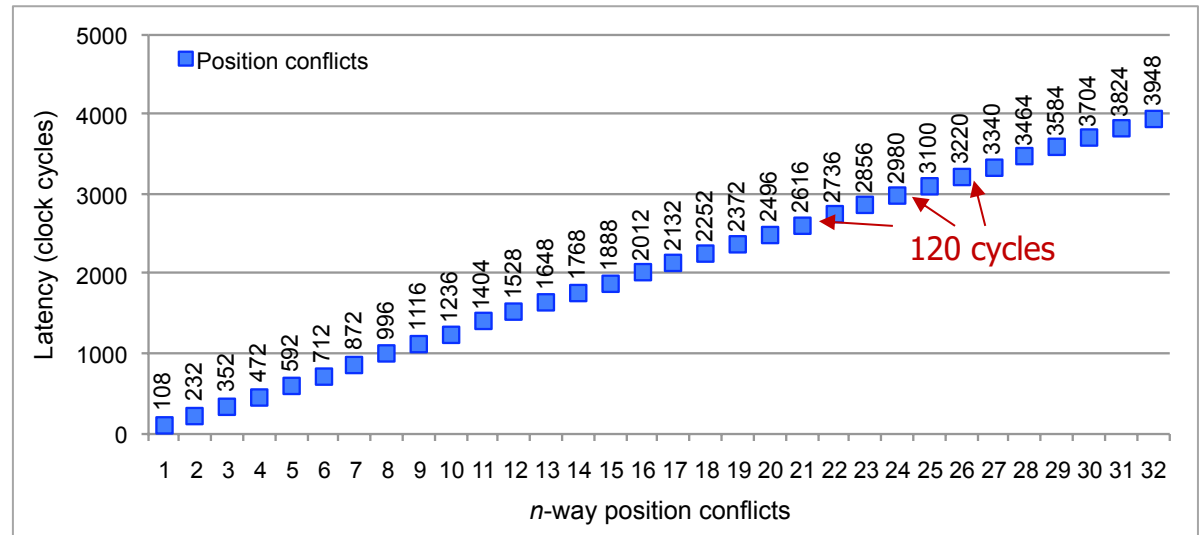
■ Microbenchmark

- Hs is a shared memory array
- clock() returns the cycle count

```
for(int i = begin; i < size; i += num_threads){  
  
    // Read from global memory  
    vote = input_data[i];  
  
    start_time = clock(); // Start timing  
  
    // Macro repeats atomicAdd 256 times  
    repeat256(atomicAdd(&Hs[vote], 1));  
  
    stop_time = clock(); // Stop timing  
}
```

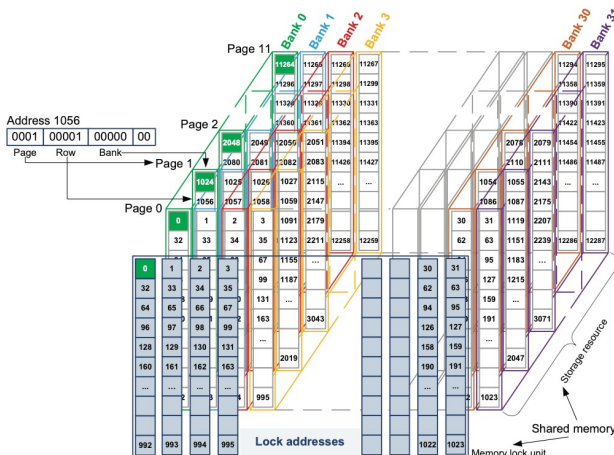
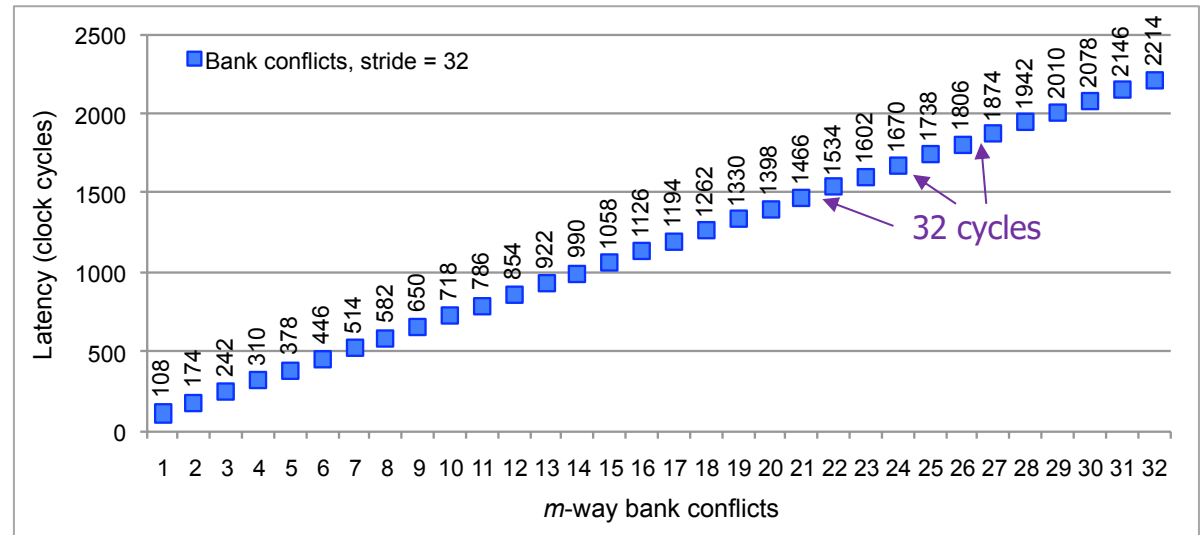
Microbenchmarking Results (I)

- **Position conflict:** n is the number of threads accessing the same address



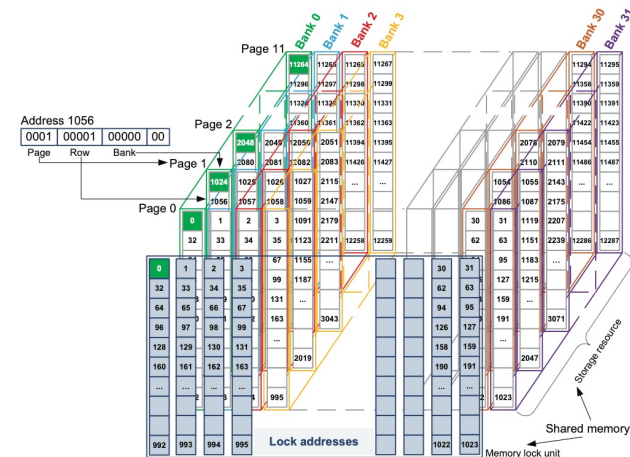
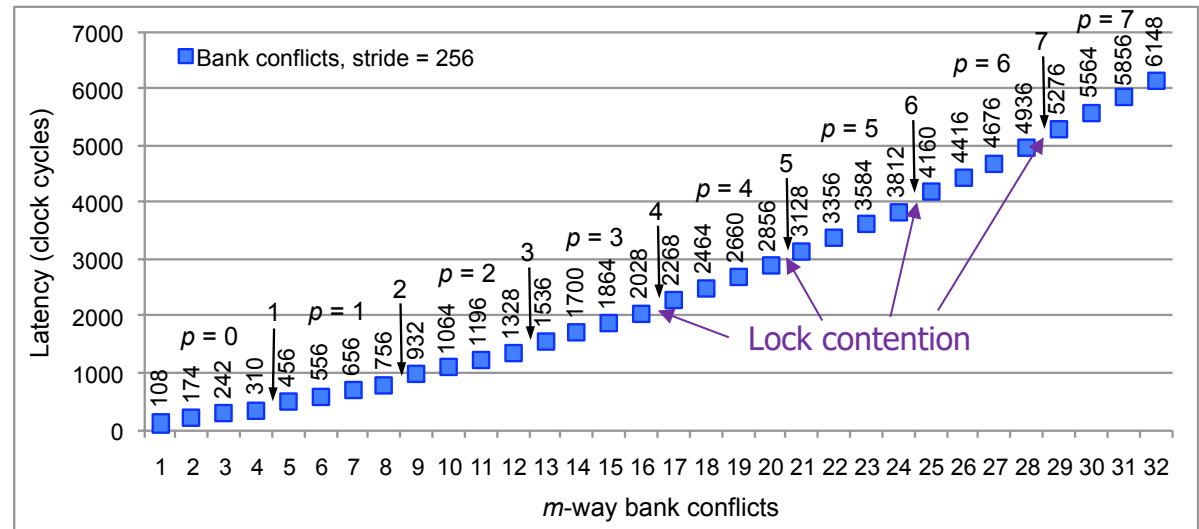
Microbenchmarking Results (II)

- **Bank conflicts:** m is the number of threads accessing the same bank (**stride 32**)



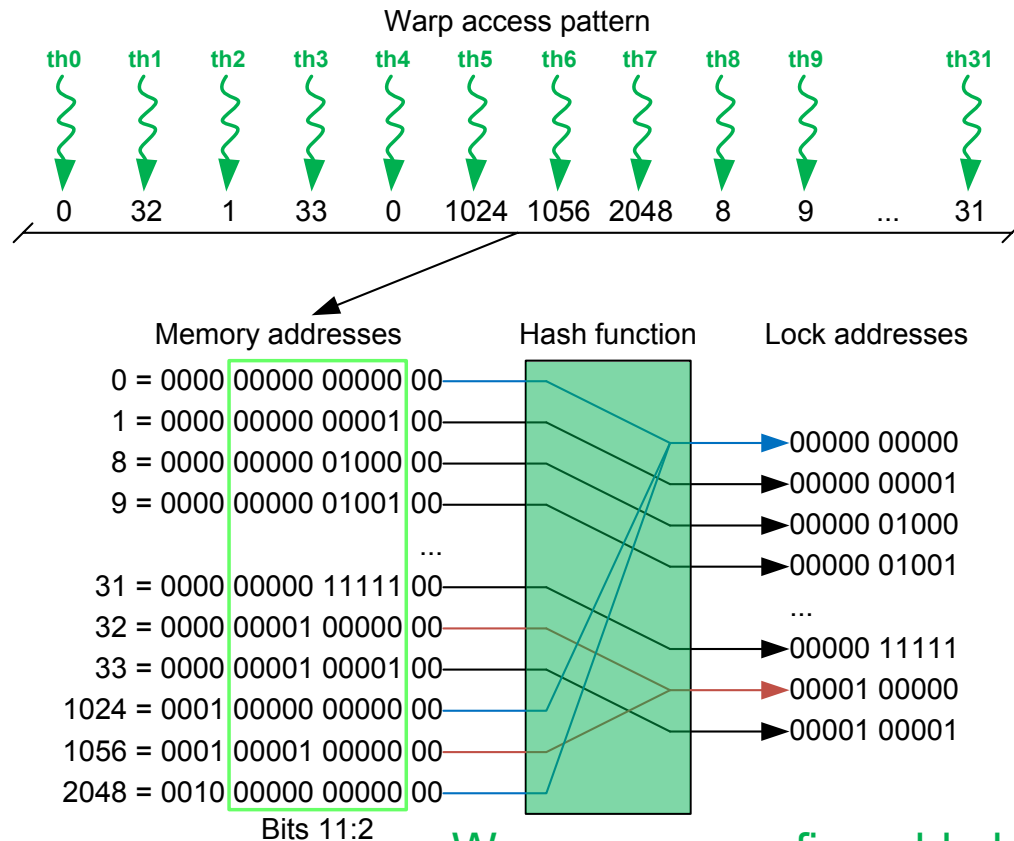
Microbenchmarking Results (III)

- Bank conflicts: m is the number of threads accessing the same bank (stride 256)



Limited Number of Lock Bits

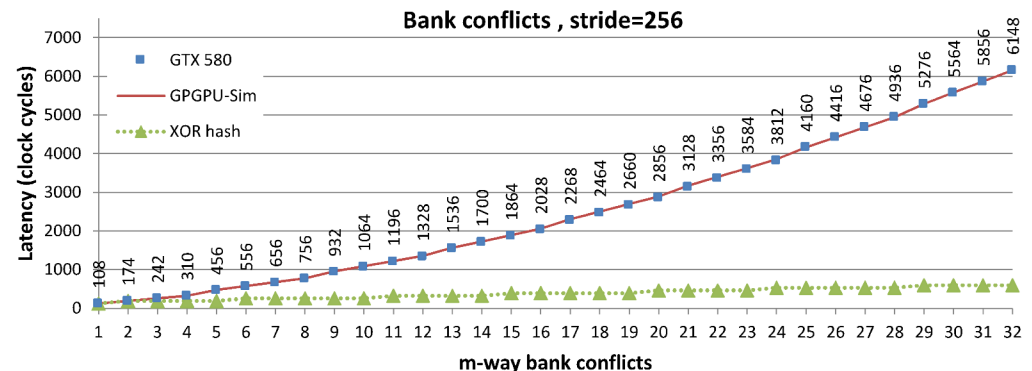
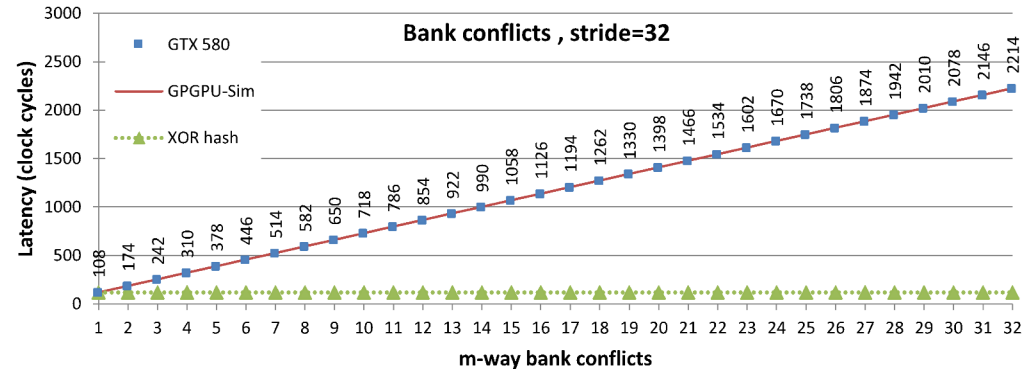
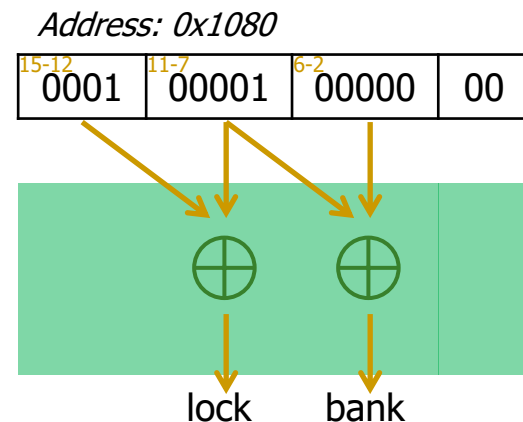
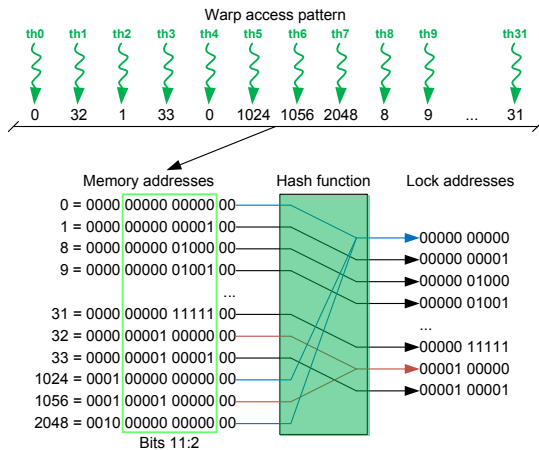
- The limited number of locks may cause **high contention**
 - 256 lock bits in Tesla, 1024 lock bits in Fermi and Kepler



We can use configurable hash functions

Configurable Hash Functions

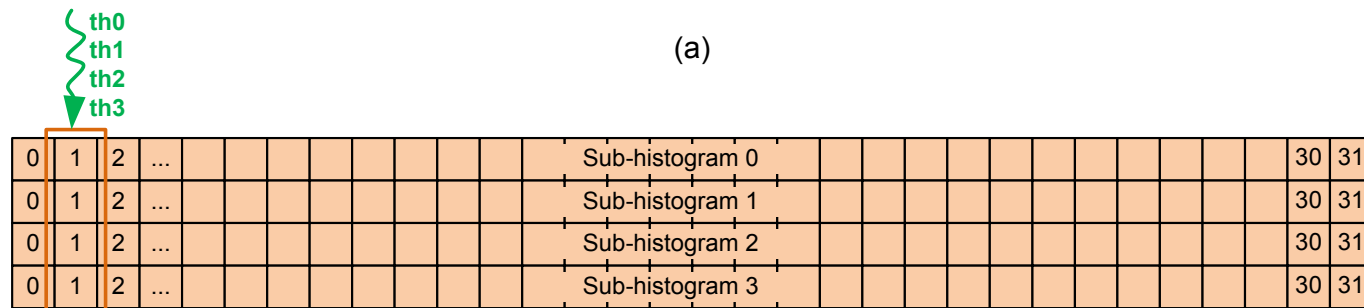
- Configurable hash functions can reduce the number of bank conflicts and lock conflicts



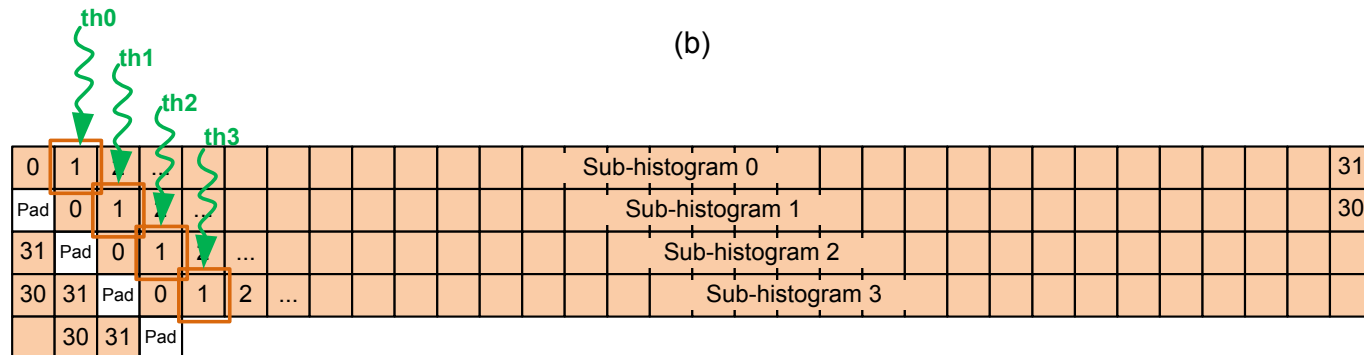
Bank and lock conflicts are greatly reduced with an XOR hash function

Optimizing Histogram Computation (I)

- Multiple private sub-histograms per block
- + padding to avoid conflicts on banks and lock bits



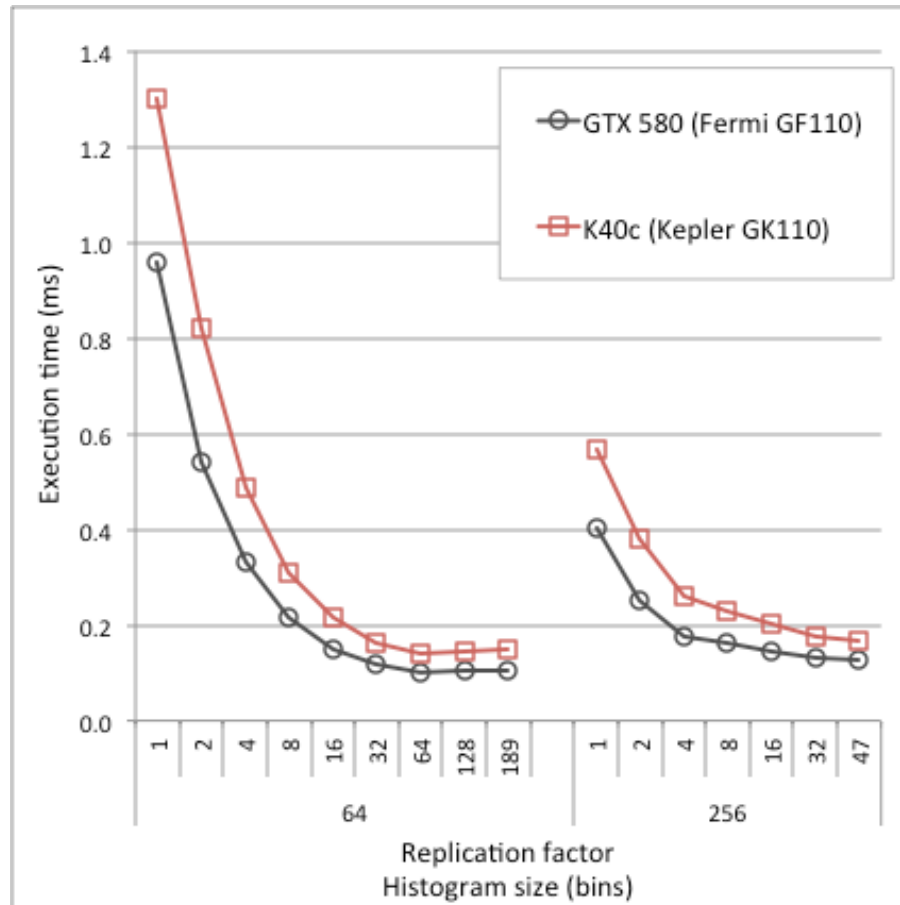
Banks 0 to 31→



Banks 0 to 31→

Optimizing Histogram Computation (II)

- Significant execution time reduction on Fermi and Kepler
 - 100 natural images



Atomic Operations on Shared Memory

- The architectural support for atomic operations evolves across GPU generations
- CUDA: `int atomicAdd(int*, int);`
- PTX: `atom.shared.add.u32 %r25, [%rd14], %r24;`
- SASS:

Tesla, Fermi, Kepler

```
/*00a0*/ LDSLK P0, R9, [R8];  
/*00a8*/ @P0 IADD R10, R9, R7;  
/*00b0*/ @P0 STSCUL P1, [R8], R10;  
/*00b8*/ @!P1 BRA 0xa0;
```

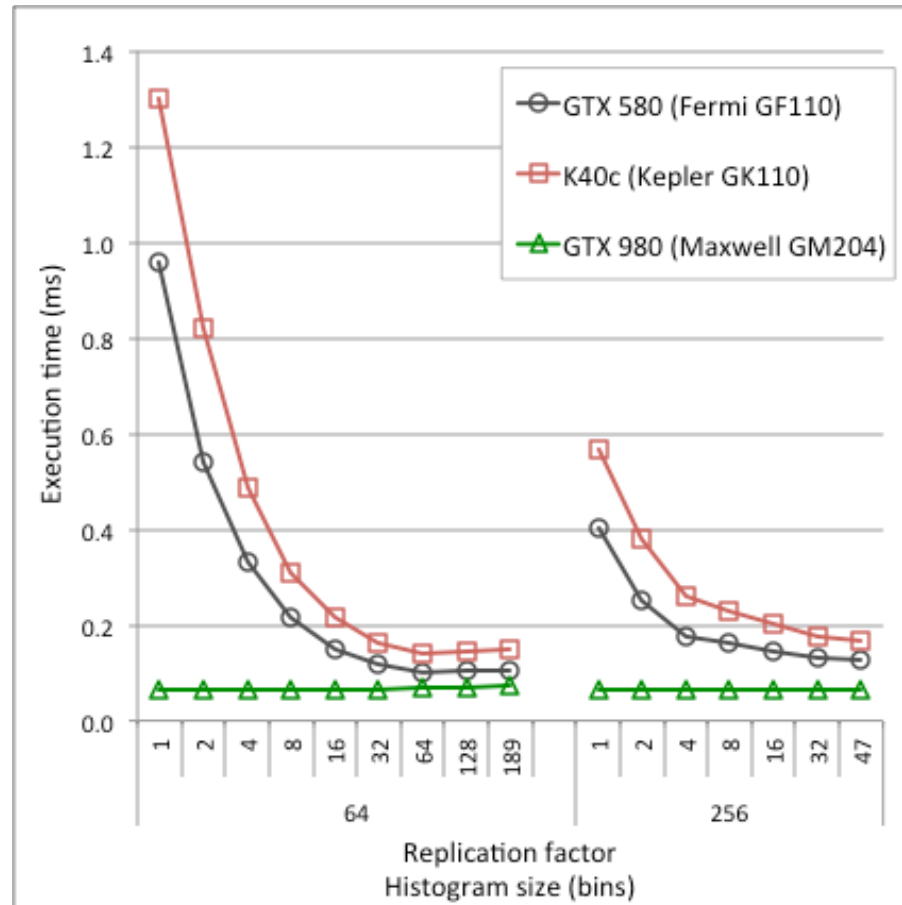
Maxwell, Pascal, Volta...

```
/*01f8*/ ATOMS.ADD RZ, [R7], R11;
```

Native atomic operations for
32-bit integer, and 32-bit and
64-bit atomicCAS

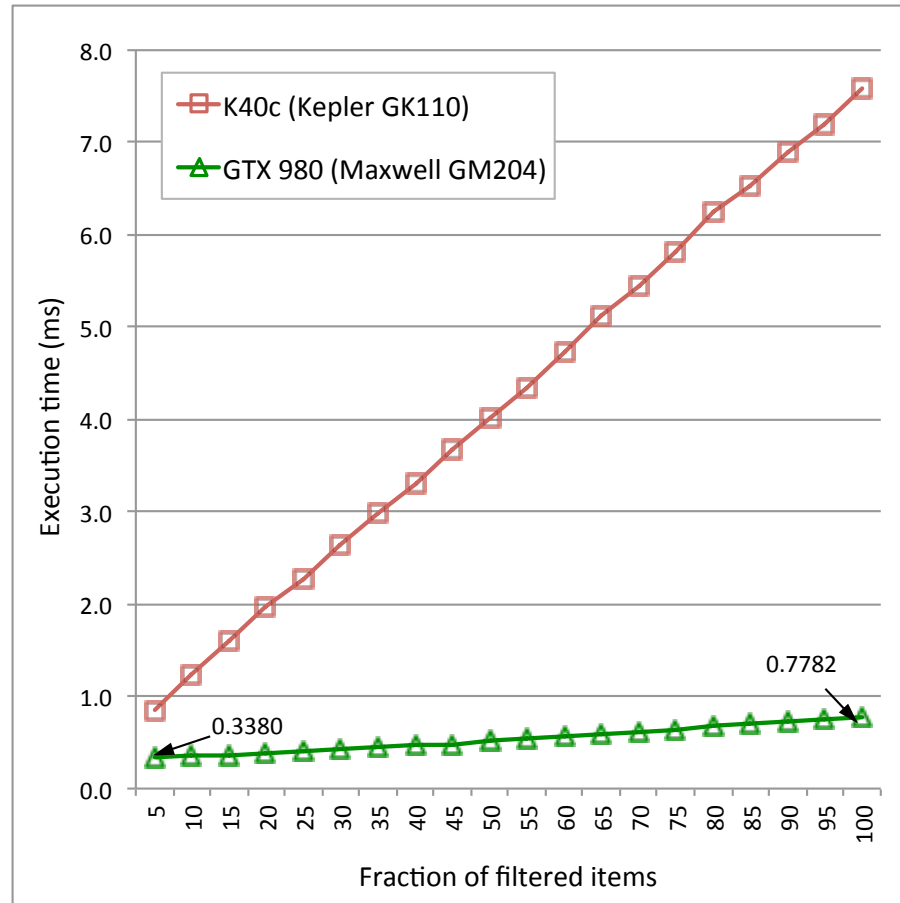
Optimizing Histogram Computation (III)

- Improved hardware (since Maxwell) saves programmers' effort



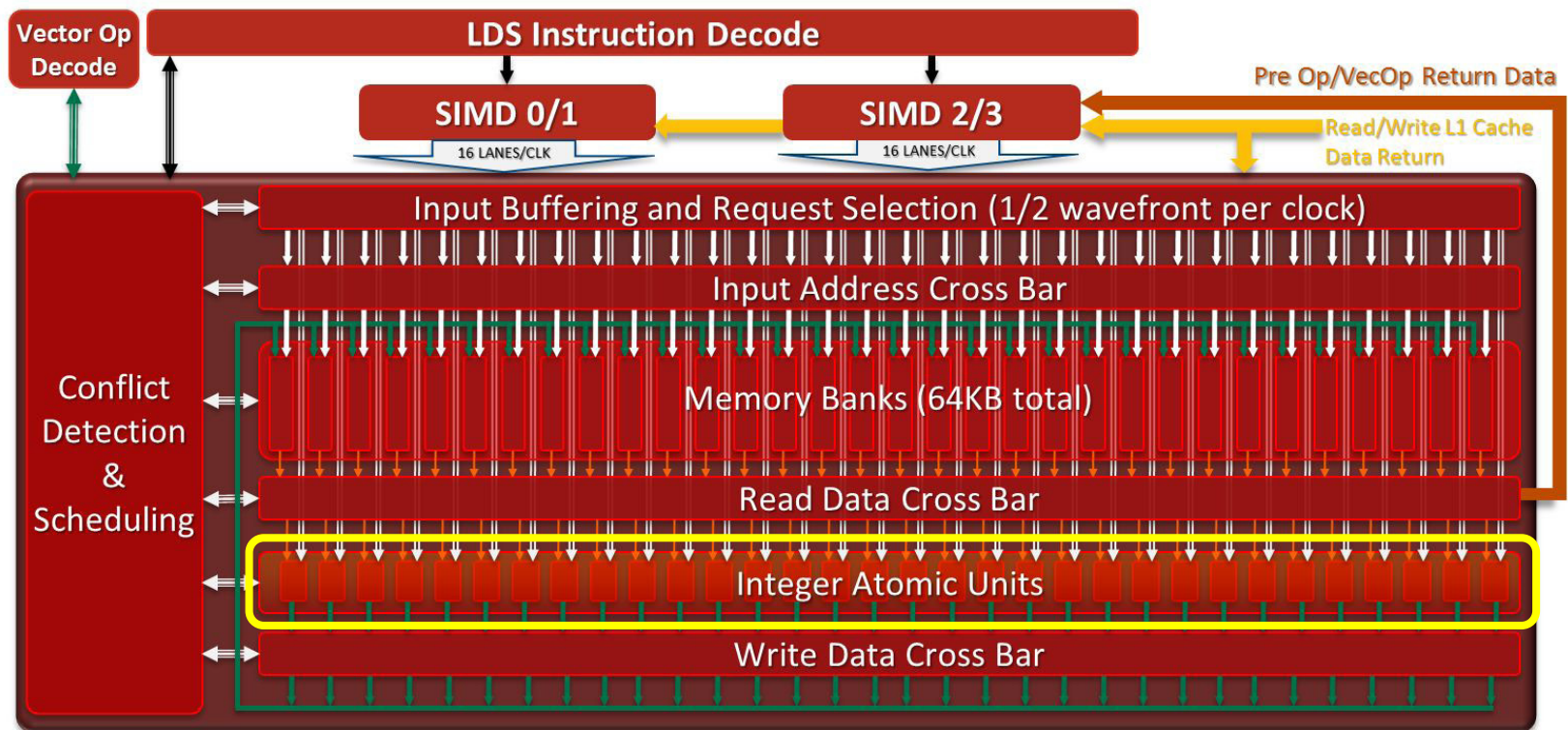
Another Example: Stream Compaction

- 1 single counter per block updated via shared memory atomics



Atomic Units Near Memory Banks

- AMD GCN architecture: Atomic units near local data share (LDS)



Atomic Operations on Global Memory

■ Tesla

- Atomic operations **executed on DRAM**

■ Fermi

- Executed on L2
- **Atomic units near L2**

■ Kepler

- Atomic units near L2 incorporate a **local buffer**

■ Pascal

- 64-bit FP atomicAdd()

Atomic Units near L2 with Local Buffer

- L2 is divided into partition units

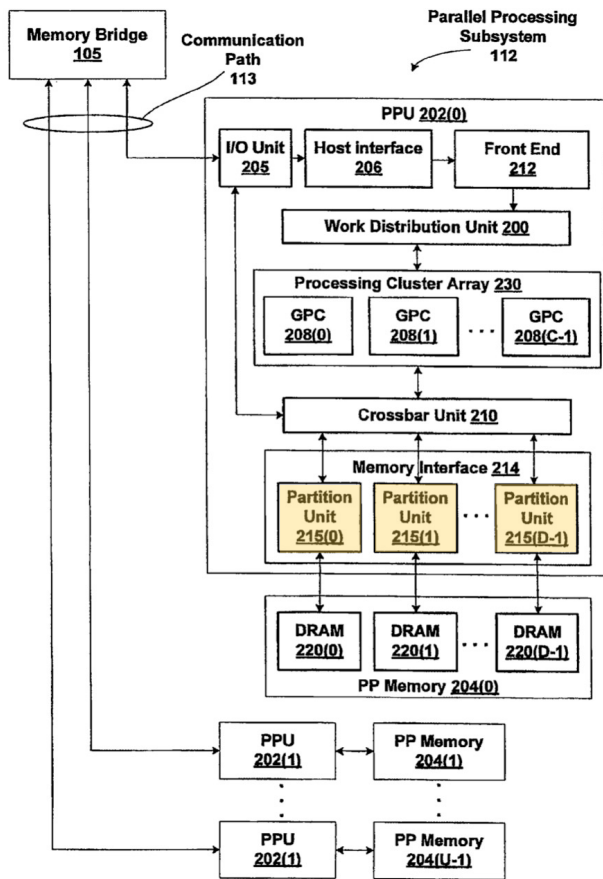
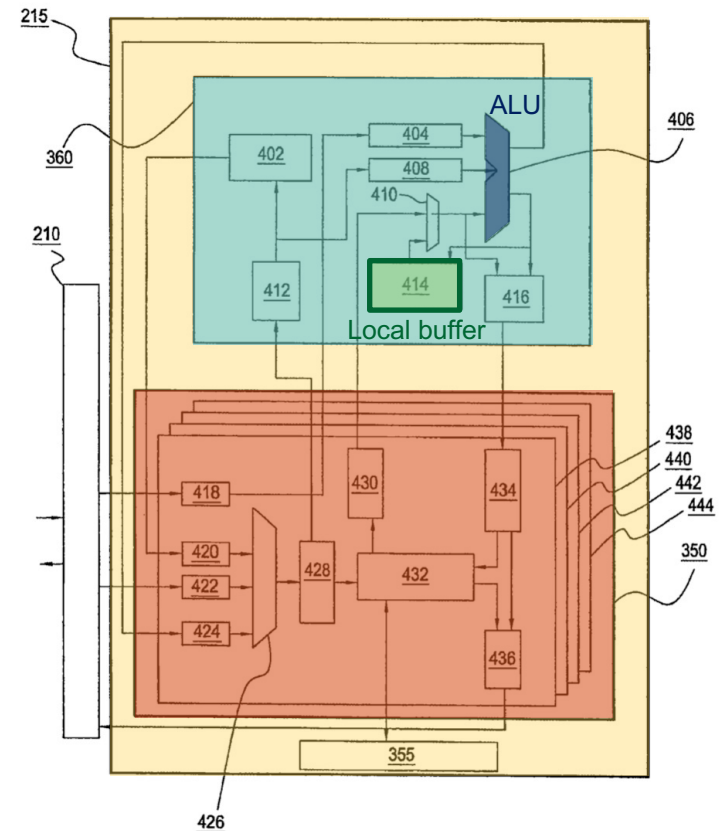
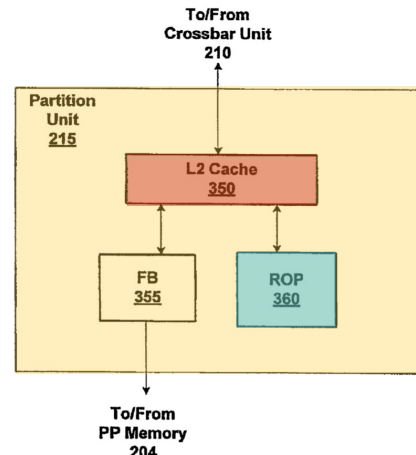
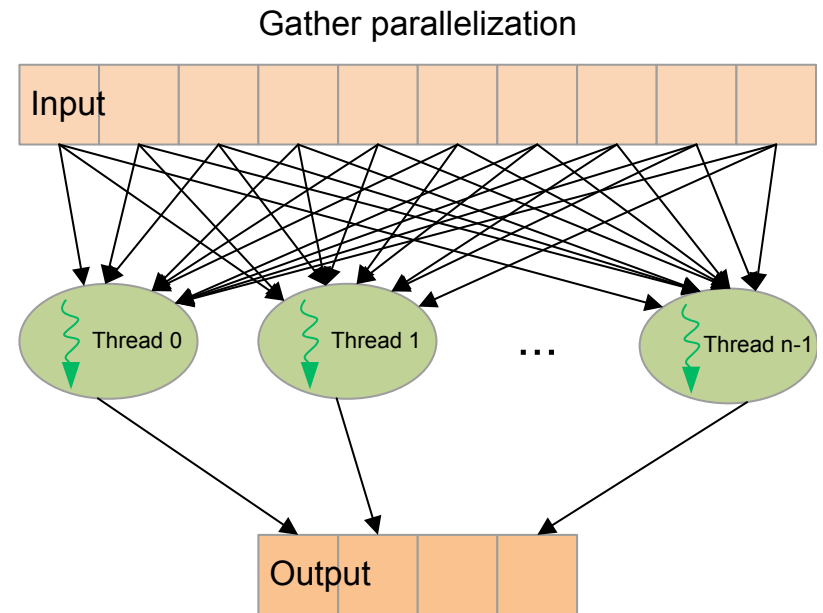
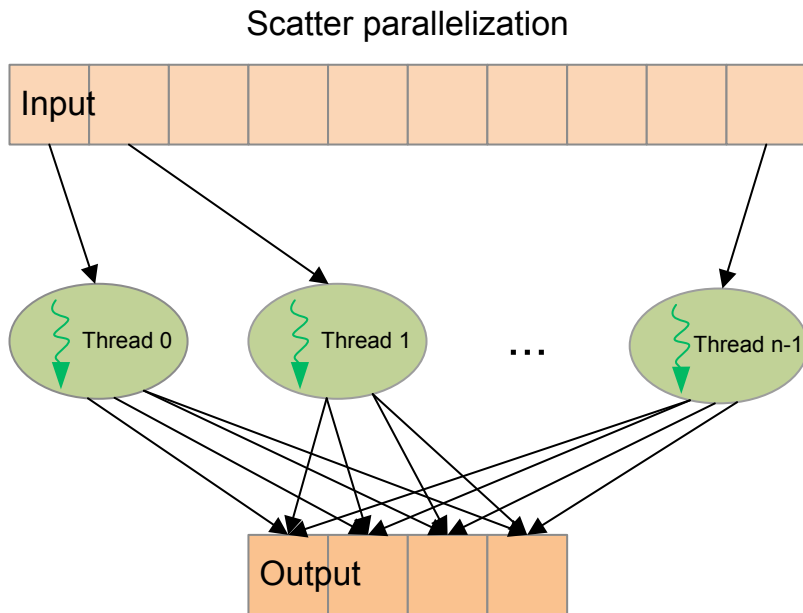


Figure 2



Scatter vs. Gather

- Scatter assigns input elements
- Gather assigns output elements



Scatter vs. Gather: Example Codes

```
__global__ void s2g_gpu_scatter_kernel(unsigned int* in, unsigned int* out,
    unsigned int num_in, unsigned int num_out) {

    unsigned int inIdx = blockIdx.x*blockDim.x + threadIdx.x;

    if(inIdx < num_in) {
        unsigned int intermediate = outInvariant(in[inIdx]);
        for(unsigned int outIdx = 0; outIdx < num_out; ++outIdx) {
            atomicAdd(&(out[outIdx]), outDependent(intermediate, inIdx, outIdx));
        }
    }
}
```

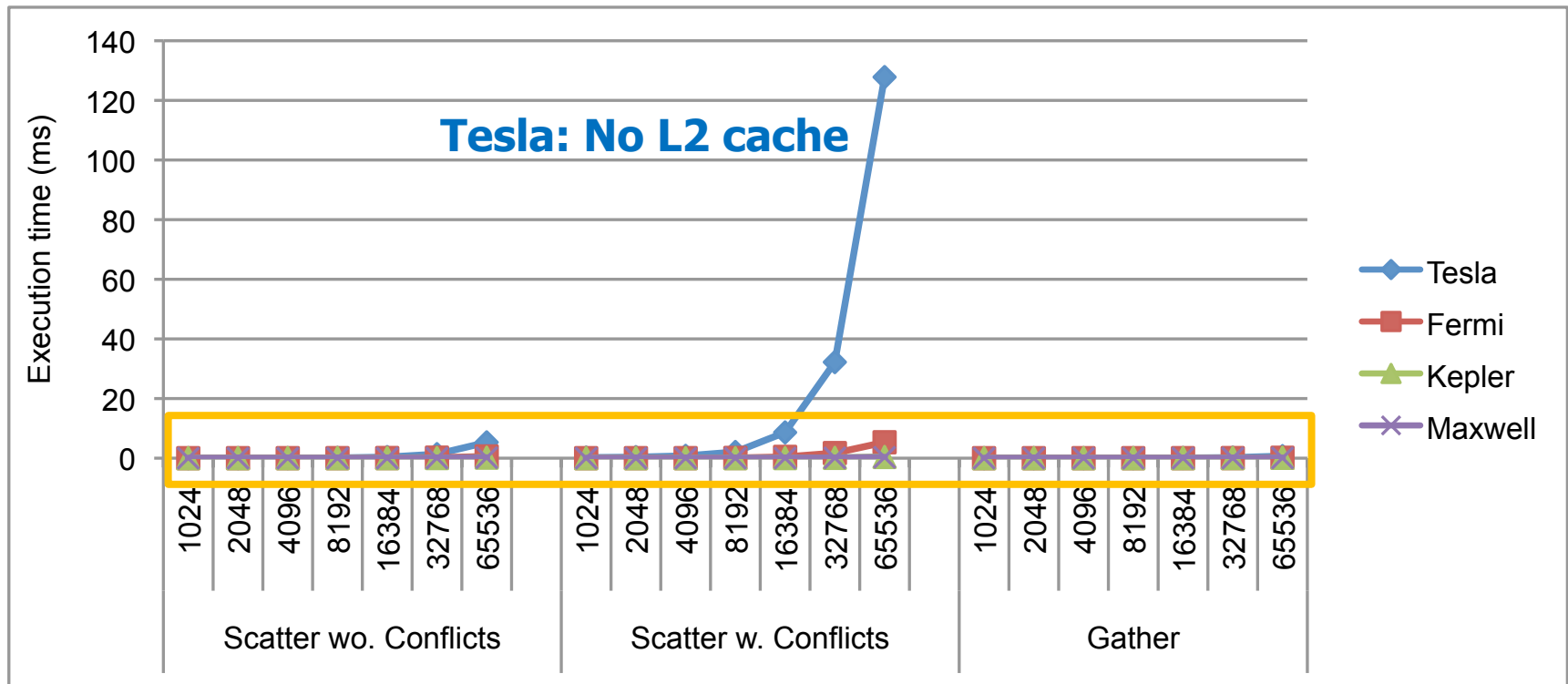
```
__global__ void s2g_gpu_gather_kernel(unsigned int* in, unsigned int* out,
    unsigned int num_in, unsigned int num_out) {

    unsigned int outIdx = blockIdx.x*blockDim.x + threadIdx.x;

    if(outIdx < num_out) {
        unsigned int out_reg = 0;
        for(unsigned int inIdx = 0; inIdx < num_in; ++inIdx) {
            unsigned int intermediate = outInvariant(in[inIdx]);
            out_reg += outDependent(intermediate, inIdx, outIdx);
        }
        out[outIdx] += out_reg;
    }
}
```

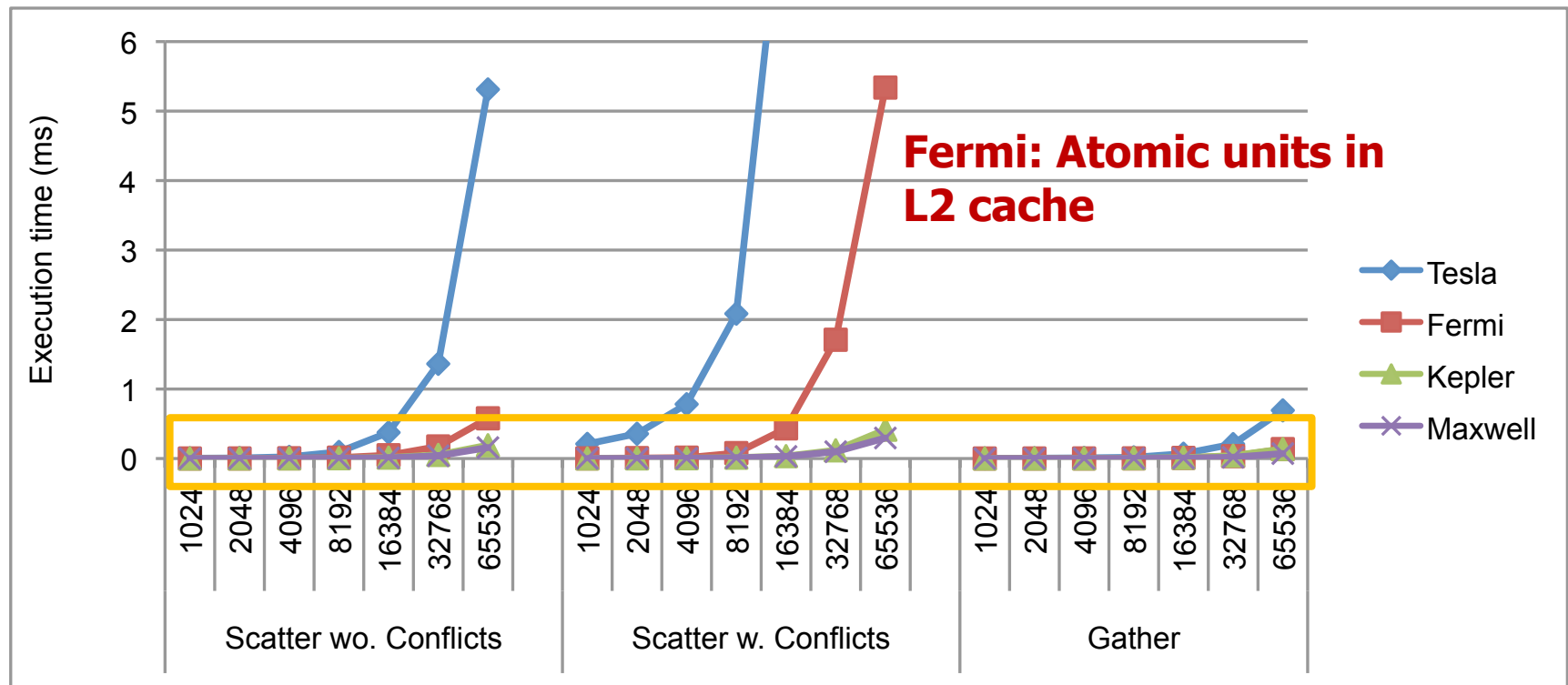

Scatter vs. Gather: Evaluation (I)

- Scatter: Large penalty due to atomic conflicts in DRAM



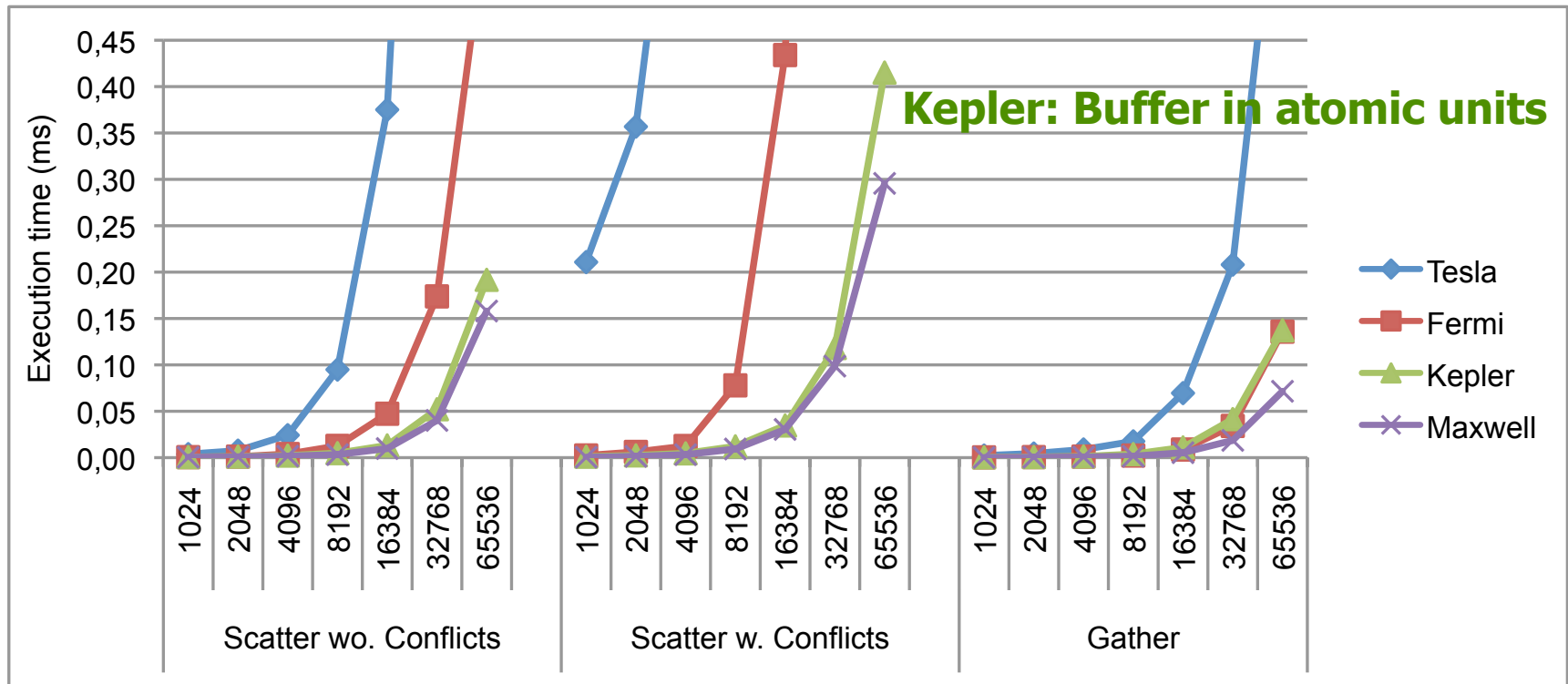
Scatter vs. Gather: Evaluation (II)

- L2 atomics improve the performance by an order of magnitude



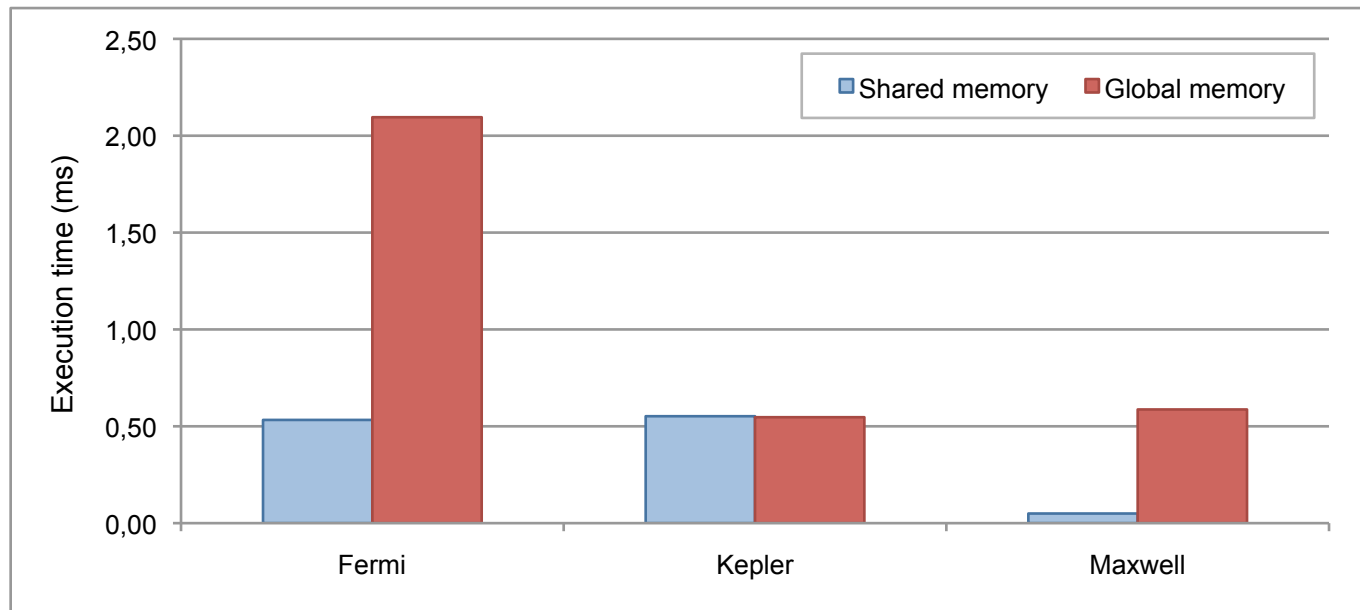
Scatter vs. Gather: Evaluation (III)

- Another 10x speedup with local buffers



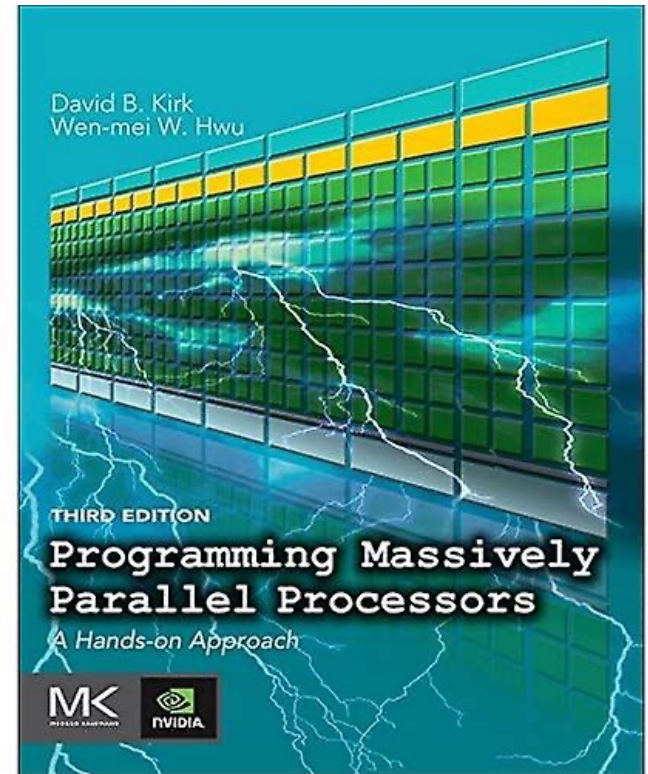
Effect of Hardware Improvements

- 256-bin histogram computation for 100 natural images
 - Shared memory implementation uses 1 private histogram per block
 - Global atomics greatly improved in Kepler
 - Native shared memory atomics since Maxwell



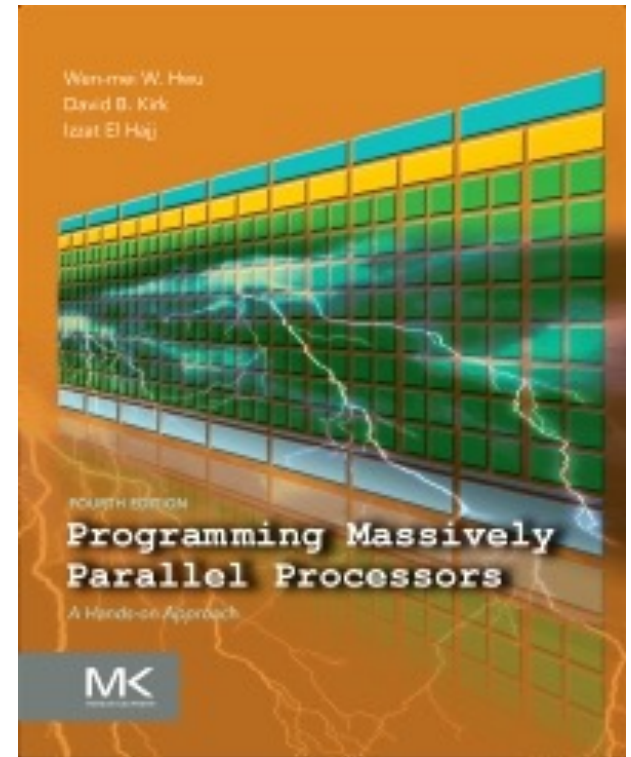
Recommended Readings (I)

- Hwu and Kirk, “**Programming Massively Parallel Processors,**” Third Edition, 2017
 - Chapter 9 - Parallel patterns — parallel histogram computation:
An introduction to atomic operations and privatization



Recommended Readings (II)

- Hwu and Kirk and El Hajj, “[Programming Massively Parallel Processors](#),” Fourth Edition, 2022
 - Chapter 9 - Parallel histogram: An introduction to atomic operations and privatization



P&S Heterogeneous Systems

Parallel Patterns: Histogram

Dr. Juan Gómez Luna

Prof. Onur Mutlu

ETH Zürich

Fall 2022

14 November 2022